Learning Apple FORTRAN

Donald J. Geenen

COMPUTER SCIENCE PRESS

Learning Apple FORTRAN

COMPUTERS AND MATH SERIES

Series Editor
MARVIN MARCUS, *University of California at Santa Barbara*

Lowell A. Carmony and Robert L. Holliday
Macintosh Pascal

Lowell A. Carmony, Robert J. McGlinn, Ann Miller Millman,and Jerry P. Becker
Apple Pascal: A Self-Study Guide for the Apple II Plus, IIe, and IIc

Lowell A. Carmony, Robert J. McGlinn, Ann Miller Millman,and Jerry P. Becker
Problem Solving in Apple Pascal

Donald J. Geenen
Learning Apple FORTRAN

Jeffrey Marcus and Marvin Marcus
Computing without Mathematics: BASIC and Pascal Applications

Marvin Marcus
Discrete Mathematics: A Computational Approach Using BASIC

Marvin Marcus
An Introduction to Pascal and Precalculus

S. Gill Williamson
Combinatorics for Computer Science

OTHER BOOKS OF INTEREST FROM COMPUTER SCIENCE PRESS

Jo Lynne Talbott Jones
The Applesoft BASIC Primer for the Apple II Plus, IIe , and IIc

W. Douglas Maurer
Apple Assembly Language

Learning Apple FORTRAN

Donald J. Geenen

Premontre High School
Green Bay, Wisconsin

COMPUTER SCIENCE PRESS

Computer Science Press, Inc.
1803 Research Boulevard
Rockville, Maryland 20850

1 2 3 4 5 6 Printing Year 91 90 89 88 87 86

Library of Congress Cataloging in Publication Data

Geenen, Donald J.
Learning Apple FORTRAN.

1. Apple IIe (Computer)—Programming. 2. Apple II Plus (Computer)—Programming. 3. FORTRAN (Computer program language) I. Title.
QA76.8.A6623G44 1985 001.64'2 84-19936

CONTENTS

To my wife,

for her faith in me.

To my students,

who wouldn't dream of letting

a bug escape unnoticed.

PREFACE

The idea for this book came to me when I was searching for a text for my high school FORTRAN class. I had recently purchased Apple FORTRAN for the school, and was surprised to find out that even though I had learned ANSI 1966 FORTRAN one year earlier, Apple's version incorporated the 1977 standard, and I would have to do some relearning.

I soon discovered that at the time—the 1981–1982 school year—there were very few texts describing the newly developed standard; worse, those that were available were definitely better suited to the advanced college-level student. What was needed was a senior high school- or college freshman-level text describing not only ANSI 1977 FORTRAN, but Apple's particular implementation of that standard. In addition, I felt the book should also contain an often overlooked topic: the operating system. After all, mastery of the operating system is equally important to mastery of a particular language. Since no textbook existed with the specifications I had in mind, I created one.

Although designed as a high school- to college-level text, this book is also well suited as a tutorial for individuals who have purchased Apple FORTRAN, since these people are given reference manuals as their sole source of documentation. Needless to say, reading and learning from them is a laborious project. In addition, these reference manuals and the system disks themselves have some "bugs," all of which are described in this book.

Before beginning, I would like to thank Dr. Marvin Marcus and Dr. Jerry Johnson for their constructive criticism. Many of their suggestions were incorporated into revisions of the manuscript, and certainly helped strengthen it.

I would also like to thank our local Apple dealer, Computer World. Their loan of a letter-quality printer allowed me to submit a professional-looking manuscript. In addition, their allowing me to preview the new Apple Pascal operating system prevented the book from being partially obsolete.

Finally, thank you, Dr. Marcus, for taking the time to read my original proposal and for recommending this book for publication. You have made a dream come true!

ORIENTATION

ORIENTATION AND OVERVIEW

A. WHAT YOU NEED TO KNOW TO USE THIS BOOK

The book assumes a previous knowledge of one computer language, preferably, though not necessarily, Applesoft BASIC. Readers are assumed to be familiar with variables, input/output commands, conditionals, looping mechanisms, arrays, functions, and subroutines, and to be at least minimally exposed to a sequential- and random-access data file system, preferably, but again not necessarily, Apple's.

The book assumes previous knowledge of neither FORTRAN nor its operating system. While the operating system is nearly identical to that used in Apple Pascal, which must be purchased in order to use Apple FORTRAN, the language sequence at our school is BASIC–FORTRAN–Pascal.

B. FORMAT OF THE TEXT

The book is divided into two major sections after the orientation: the operating system and the FORTRAN language itself. Each of these two sections is then subdivided into chapters. Chapters generally begin with a formal description of the topic under consideration—explaining prompts, command syntax, and so forth. In order to differentiate between prompts and user replies, all user replies appear in **emphasized print.** The description is followed by several examples that illustrate actual use. Each chapter then closes with a set of review questions, as well as a set of "hands-on" exercises.

At the end of the book are two appendices. Appendix A contains answers to all the review questions, while Appendix B explains the rather cryptic error messages of Apple FORTRAN in a more detailed, yet more lucid form that that of the Language Reference Manual.

Finally, a disk to supplement the text is available. Its name is LAF (not because it's funny, but because it stands for *Learning Apple FORTRAN;* clever, huh?). This disk contains text versions of all the sample programs in the book, and for many of the longer and more elaborate programs, the code (or compiled) versions as well.

C. WHERE SHOULD YOU BEGIN READING?

The answer depends to a large extent on your previous experience, of course, but I will suggest the following "road maps" for various readers:

All readers: Read the remainder of this introduction to learn your equipment needs, and to get your system disks configured correctly. I also suggest reading Chapter 20 any time after reading Chapters 1 and 2. You will probably be surprised to learn that you have a word processor at your service for no extra charge!

Readers with no previous exposure to FORTRAN or the Apple Pascal operating system: As you might guess, you need to start at Chapter 1, and proceed through each consecutive chapter. Please take your time when covering the first five chapters. The importance of mastering the operating system cannot be overemphasized. Chapters 15 and 21 are optional material for your purposes. These chapters are largely supplementary, so their omission will not hinder you in later chapters. You may also choose to cover Chapters 18 and 19, which focus on graphics, sound, and so forth, before covering Chapters 16 and 17, which focus on data files. Again, these chapters are structured such that you may read them out of sequence without losing a sense of continuity.

Readers with no previous exposure to FORTRAN, but who are familiar with the Apple Pascal operating system: Begin by reading the remainder of this introduction, for it will help you pick up some operating system qualities unique to Apple FORTRAN. You may omit Chapters 1 and 2, for the Editor and Filer have not been altered in any way. Continue with Chapters 3, 4, and 5, where you will learn that there are three Compiler prompts to be answered (as opposed to two in Pascal), and that every FORTRAN program must be linked. Once you've picked up the operating system "quirks," you may begin with the FORTRAN language in Chapter 6 and proceed through consecutive chapters, keeping in mind that, as a FORTRAN novice, you may not want to cover Chapters 15 and 21 your first time through. Likewise, you may also elect to read Chapters 18 and 19 before reading Chapters 16 and 17 (as mentioned above).

Readers who are familiar with FORTRAN, but who have had no exposure to the Apple Pascal operating system: Avoid the temptation to dive into the language without first mastering the operating system! Read Chapters 1 through 5 carefully and thoroughly. Next, you will be glad to know that Apple FORTRAN conforms to ANSI 1977 Standard subset FORTRAN. As a result, you should be able to skim Chapters 6 through 14.

NOTE: If you have been using a 1966 compiler, you should read Appendix F in the Language Reference Manual for a synopsis of 1977 vs. 1966 standard FORTRAN. You may want to spend some time with Chapter 6 and, especially, Chapter 9.

Chapters 15, 16, and 17 should be of great interest to you, for they describe Apple FORTRAN's method for creating library files and data files. Finally, Chapters 18, 19, and 21 cover Apple's extensions for reading game paddles, generating random numbers, sound, and graphics.

D. EQUIPMENT YOU WILL NEED

You will need the standard 64K Apple IIe or Apple IIc computer (or a 48K Apple II Plus computer with an additional 16K language card), some sort of monitor device, and a printer. (The printer is optional, but highly recommended.) You will need to own Apple FORTRAN, as well as Apple Pascal because Apple FORTRAN uses the Apple Pascal operating system. Finally, you will need two disk drives.

NOTE: This book has been written assuming a two-disk-drive setup.

Although Apple FORTRAN can be run on a system with only one drive, such opeation is very tedious and fails to take full advantage of the power of the Apple FORTRAN operating system. Those blessed with three or more drives can also benefit from this book. These users will be able to place the two system disks in Drives 1 and 2, their own disk in Drive 3, and then proceed to ignore all references to disk shuffling.

E. CONFIGURING YOUR SYSTEM DISKS

Apple FORTRAN as purchased is incomplete. The two system disks (named FORT1 and FORT2) contain two files each. Before you can get FORTRAN up and running, you must transfer the entire operating system (seven files) as well as two utility files from the Apple Pascal system disks to the Apple FORTRAN system disks. Finally, you must add a newly written file to disk FORT1 to compensate for a "bug."

The particular arrangement of these ten extra files on the system disks is called the *configuration.* The FORTRAN disks may be configured in several different ways, but note the following:

NOTE: Do not use the disk configuration described by the Apple FORTRAN Language Reference Manual.

The setup given in the reference manual unnecessarily duplicates files on both system disks, and leaves precious little room for intermediate versions of your developing program.

This *alternate* configuration leaves 117 free blocks on your boot diskette,

rather than the reference manual's suggested 40. In addition, the disk containing the valuable FORTRAN Compiler will be write-protected. Write-protection is something you could not have in using Apple's configuration unless you sacrificed a valuable system shortcut. Here is the setup:

FORT1	FORT2
FORTLIB.CODE	SYSTEM.COMPILER
FORTMOD.CODE*	SYSTEM.EDITOR
SYSTEM.LIBRARY	SYSTEM.FILER
SYSTEM.APPLE	SYSTEM.LINKER
SYSTEM.PASCAL	FORMATTER.CODE
SYSTEM.MISCINFO	FORMATTER.DATA
SYSTEM.CHARSET	
SYSTEM.STARTUP	
117 blocks free	73 blocks free

*not present in all versions of Apple FORTRAN

F. ARRIVING AT THE NEW CONFIGURATION

This section takes you through the steps necessary to configure your system disks in the manner described above. Don't panic because of the length of the explanation. It is a one-time event, and the directions are written for complete novices. It will take anywhere from 35 minutes (if you are a fast reader/operator and make no mistakes) to 1 hour and 15 minutes (if you are a computer novice and need to take things very slowly).

If you'd rather know something about the operations we will be performing (as opposed to being led blindly through them), you may first refer to Chapter 2, which describes the operating system's Filer. Read the sections describing the options REMOVE, TRANSFER, and LIST DIRECTORY.

Before we begin, you should have within reach:

1. Apple FORTRAN disks FORT1 and FORT2, as well as the supplied FORT2 backup disk. (With a felt-tipped pen, write the word "backup" on the disk's label so you don't get it confused with the original.)

2. Apple Pascal disks APPLE1, APPLE2, and APPLE3.

IMPORTANT NOTE: APPLE1 must not be write-protected! If your APPLE1 disk does not have a small rectangular hole about one inch down from the top right side as you look at its label, you need to make a non-write-protected backup copy of the disk. See Appendix E of the Apple Pascal Language Reference Manual *to do so.*

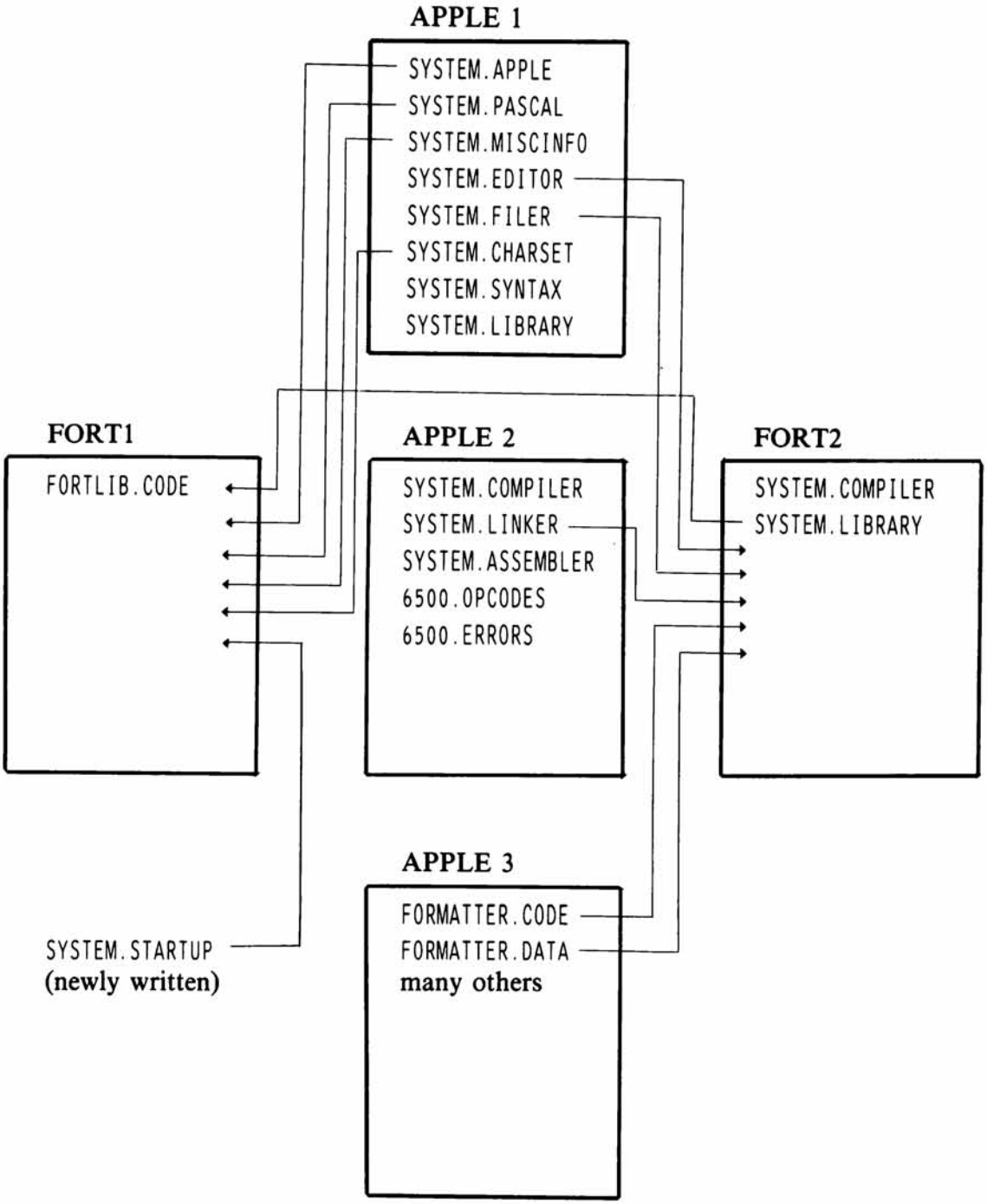


Figure 1.1 Apple FORTRAN Disk Configuration: The Game Plan

3. One blank diskette, to be used as a backup disk for FORT1.

Step 1: Take disks APPLE1 and APPLE2 out of their sleeves. Place disk APPLE1 label-side up into Disk Drive 1. Close Drive 1's door. Place disk APPLE2 into Drive 2, and close its door also. Turn your monitor (CRT) on. Turn on the computer with the switch located in the left rear of the machine. After about 15 seconds, you will see a greeting page ("`Welcome APPLE1 to Apple II Pascal. . .`," etc.). You have now successfully "booted" Pascal! This greeting page also contains a menu of options known as *Command Level* ("`COMMAND: E(DIT, R(UN, F(ILE, C(OMP. . .`," etc.).

Step 2: Press the **"F"** key. After Drive 1 spins for 3 seconds or so, you will be informed that you are in the Filer. Replace disk APPLE1 with disk FORT1. (Don't forget to shut the drive's door!) Also replace disk APPLE2 with disk FORT2.

You may skip directly to Step 8 if your FORTRAN system disks have just been purchased and are fresh out of the wrapper. Steps 3 through 7 serve to "undo" those disks that have already been configured according to the Language Reference Manual's suggestion. If you aren't sure about the status of your disks, continue with Step 3.

Step 3: Press the **"R"** key. You will see the prompt "REMOVE?" appear when you have done so. Answer by typing "FORT1:?" *(NOTE: Do NOT type the quotation marks!)* The screen should look like this (with your answer in **emphasized print**):

```
REMOVE ? FORT1:?
```

If you make a mistake, simply use the left-arrow key to back up to your mistake, then re-type.

When the screen looks as it does above, **press the "RETURN" key.**

A common error is to accidentally omit the "?" after the "FORT1:". If you do so, you will be asked if you want to "DESTROY FORT1:?". Your obvious answer is **"N"**, after which you'll have to start Step 3 over.

Another common error is to misspell the answer to the prompt. If you do, you will receive the message "VOL NOT ON-LINE" or "FILE NOT FOUND". To recover, simply start over at the beginning of Step 3.

Step 4: You will now receive a "REMOVE?" prompt for each of the files contained on disk FORT1. *You should type an "N" (no quotes) for the files FORTMOD.CODE and FORTLIB.CODE, and type a "Y" for all others (if there are any).* Here is a typical run-through of Step 4 (the files may be present in a different order):

```
REMOVE FORTMOD.CODE ? N
REMOVE FORTLIB.CODE ? N
REMOVE SYSTEM.APPLE ? Y
REMOVE SYSTEM.PASCAL ? Y
REMOVE SYSTEM.MISCINFO ? Y
REMOVE SYSTEM.CHARSET ? Y
REMOVE SYSTEM.FILER ? Y
REMOVE SYSTEM.LINKER ? Y
REMOVE SYSTEM.EDITOR ? Y
```

NOTE: Some people will have files to remove, others will not. It depends on whether or not the disks were already configured according to the Reference Manual's recommendation. It is also possible that some people will not have

file FORTMOD.CODE present. If you do not, don't panic, for it is unnecessary in normal use.

At the end of this process, you are asked whether or not you want to "UPDATE DIRECTORY?." *If you have answered any of the prompts incorrectly, type "N" and return to Step 3.* Otherwise, if all is OK, type **"Y"** as follows:

```
UPDATE DIRECTORY ? Y
```

Step 5: Press the **"R"** key once again. This time, reply with "FORT2:?" and press the **"RETURN"** button:

```
REMOVE ? FORT2:?
```

Step 6: Type an **"N"** for the files SYSTEM.COMPILER and SYSTEM.LIBRARY, and a **"Y"** for all the others. Then, when asked whether or not to "UPDATE DIRECTORY?", type **"Y"** if you've made no mistakes. *If you accidentally typed the wrong key for any of the files, type "N" and return to Step 5.* A typical application follows:

```
REMOVE SYSTEM.COMPILER ? N
REMOVE SYSTEM.LIBRARY ? N
REMOVE SYSTEM.APPLE ? Y
REMOVE SYSTEM.PASCAL ? Y
REMOVE SYSTEM.MISCINFO ? Y
REMOVE SYSTEM.CHARSET ? Y

UPDATE DIRECTORY ? Y
```

Step 7: You don't have to do any work for this step. We're simply going to take a breather and explain what we've done. After all this work, we simply have FORT1 and FORT2 back in their original, as-purchased setup. FORT1 contains files FORTLIB.CODE and, with some versions, FORTMOD.CODE. FORT2 contains the files SYSTEM.COMPILER and SYSTEM.LIBRARY. *If you have any doubt as to whether or not this is true of your disks, it would be wise to repeat Steps 3 through 6, just to be sure.*

Now that the system disks are back in their original form, the remaining steps will serve to set them up in the new, more efficient configuration. All set? OK, here we go!

Step 8: Type the **"T"** key. When the machine asks you what to "TRANSFER?", respond with "FORT2:SYSTEM.LIBRARY" and then press **RETURN.** When asked "TO WHERE?", respond "FORT1:$", as follows (if you're wondering, the "$" is shorthand for "same name"):

```
TRANSFER ? FORT2:SYSTEM.LIBRARY
TO WHERE ? FORT1:$
```

Again, a common error is to misspell an answer. If you see a "FILE NOT FOUND" or "VOL NOT ON-LINE" in this or other steps, just start the step over from the beginning, and be more careful the second time through.

Step 9: Press the **"R"** key, and respond as shown in the **emphasized print** below. Recall that whenever you remove a file, the operating system will ask you whether or not it should "UPDATE DIRECTORY?". Unless you made a mistake, type **"Y"**:

```
REMOVE ? FORT2:SYSTEM.LIBRARY
FORT2:SYSTEM.LIBRARY --> REMOVED
UPDATE DIRECTORY ? Y
```

Step 10: Remove disk FORT2 from Drive 2 and replace it with disk APPLE1. (FORT1 should still be in Drive 1.) Type the **"T"** key, then answer the transfer prompts as shown below (you will need to press the **"RETURN"** button after each of your two replies):

```
TRANSFER ? APPLE1:?
TO WHERE ? FORT1:$
```

You will now receive a list of all APPLE1 files, and be asked whether or not they should be transferred. You will respond with **"Y"** for files SYSTEM.APPLE, SYSTEM.PASCAL, SYSTEM.MISCINFO, and SYSTEM.CHARSET, as shown below:

```
TRANSFER SYSTEM.APPLE ? Y
TRANSFER SYSTEM.PASCAL ? Y
TRANSFER SYSTEM.MISCINFO ? Y
TRANSFER SYSTEM.EDITOR ? N
TRANSFER SYSTEM.FILER ? N
TRANSFER SYSTEM.LIBRARY ? N
TRANSFER SYSTEM.CHARSET ? Y
TRANSFER SYSTEM.SYNTAX ? N
```

Step 11: You should now have disk FORT1 nearly set, with the sole exception of file SYSTEM.STARTUP. To be sure, type **"L"**. Respond to the prompt as shown below (don't forget the colon!):

```
DIR LISTING OF ? FORT1:
```

You should now see the files listed earlier for the new configuration (see Section E, *"Configuring Your System Disks"*; the order in which the files appear makes no difference whatsoever). If any files except SYSTEM.STARTUP are missing, you may have to repeat Step 10. If you have

any extra files, return to Step 3 to remove them (note that this time through Step 3, you should remove only the extra files you have found). After repeating either Step 3 or Step 10, come right back to Step 11 to verify your results. Now we're ready to operate on disk FORT2.

Step 12: Remove disk FORT1 from Drive 1 and replace it with disk FORT2 (leave APPLE1 in Drive 2). Your task is to transfer files SYSTEM.EDITOR and SYSTEM.FILER from APPLE1 to FORT2. After you press **"T"** for "TRANSFER", the rest of the sequence is given below:

```
TRANSFER ? APPLE1:?
TO WHERE ? FORT2:$

TRANSFER SYSTEM.APPLE ? N
TRANSFER SYSTEM.PASCAL ? N
TRANSFER SYSTEM.MISCINFO ? N
TRANSFER SYSTEM.EDITOR ? Y
TRANSFER SYSTEM.FILER ? Y
TRANSFER SYSTEM.LIBRARY ? N
TRANSFER SYSTEM.CHARSET ? N
TRANSFER SYSTEM.SYNTAX ? N
```

Step 13: Remove disk APPLE1 from Drive 2 and replace it with disk APPLE2. (FORT2 should still be in Drive 1.) You will be transferring a single file from APPLE2 to FORT2. Type **"T"**, then answer the prompts:

```
TRANSFER ? APPLE2:SYSTEM.LINKER
TO WHERE ? FORT2:$
```

Notice that this time you will *not* have to answer "yes" or "no" for a list of all files. The same is true of the next step.

Step 14: Remove disk APPLE2 from Drive 2 and replace it with disk APPLE3. Press **"T"**, then proceed as follows:

```
TRANSFER ? APPLE3:FORMATTER=
TO WHERE ? FORT2:$
```

This will actually transfer *two* files, for "=" is a wildcard symbol.

Step 15: FORT2 should now be configured correctly! Let's make sure, though. Press **"L"**, then answer as shown (again, don't forget the colon):

```
DIR LISTING OF ? FORT2:
```

If your directory listing matches that shown earlier (see Section E, *"Configuring Your System Disks"*), you're finished with FORT2. If you are missing

files, return to Step 12, 13, or 14. If you have extras, repeat Step 5. When finished, come back to Step 15.

Step 16: We're now going to do a little preventive maintenance. Take FORT2 out of Drive 1 and carefully place a small piece of tape over the write-protect opening (the small rectangular cut-out about one inch down from the upper right of the disk as its label faces you). This prevents accidental erasure of the precious SYSTEM.COMPILER file, of which you will be unable to make backup copies. In fact, the protection feature won't allow removal of any FORT2 files, nor will it allow you to save any files on the disk.

Place FORT2 in a safe place for about 10 minutes, for we still need to make a backup copy. First, though, we need to add one more file to FORT1.

Step 17: Remove APPLE3 from Drive 2. Place APPLE1 in Drive 1 and APPLE2 in Drive 2. Press **"Q"** to quit the Filer. Command Level should now reappear ("COMMAND: E(DIT,R(UN,F(ILE,C(OMP. . .", etc). Your mission (should you choose to accept it) is to write a Pascal program to compensate for a bug in the FORTRAN system disks. Without this file, you could never access the graphics, sound, game paddle, and random number generator routines. Let's begin!

Step 18: Press **"E"** for Editor. Drive 1 will spin, and you'll see a prompt "NO WORKFILE IS PRESENT. . . .", etc. Press the **"RETURN"** key.

NOTE: If you DIDN'T see the prompt, you may recover by typing the seven keys (no quotes) "Q E F N Y Q E", then pressing "RETURN" as mentioned above.

You should now see the Editor prompt line ("EDIT A(DJST C(OPY D(LETE. . .", etc.).

Step 19: Type **"I"** for INSERT (a new prompt line will appear). Now type the following four-line program. Press **"RETURN"** at the end of each line:

```
PROGRAM INITFORTRAN;
USES APPLESTUFF;
BEGIN
END.
```

If you make any errors, just back up with the left-arrow key and try again. When it looks *exactly* like the above program (same spelling, punctuation, etc.), press **CTL-C** (i.e., hold down the "CONTROL" or "CTRL" button, then press "C" *at the same time*). When you do so, the Editor prompt line will reappear.

Step 20: Now we are going to translate that program into a type of machine language. To do so press three keys: **"QUC".** This process will take about 45 seconds.

If you receive the error message "ERROR WRITING OUT THE FILE", *it means your APPLE1 disk is write-protected. You must make a non-write-protected backup copy (the procedure is given in Appendix E of the* Apple Pascal Language Reference Manual*) and then begin again at Step 18 with your NEW APPLE1 in Drive 1.*

If you receive any other error message, press "E". When the file is read into the Editor, press the spacebar, then return to Step 18's NOTE to recover. You must then re-do Steps 19 and 20.

Step 21: You'll be glad to know we're almost home! The file you've just created must be transferred to disk FORT1. To begin, type **"F"** for Filer. After the Filer has been read in, remove disk APPLE2 from Drive 2, and replace it with disk FORT1.

Step 22: Press **"T"** and answer the following prompts as shown (it is crucial to spell the second answer correctly.):

```
TRANSFER ? SYSTEM.WRK.CODE
TO WHERE ? FORT1:SYSTEM.STARTUP
```

Now, you were shown in Step 11 how to get a directory listing of FORT1. Return to that step, but this time expect to find *all* of the files listed for FORT1. After your brief detour to 11, pick up at Step 23, where we'll make backup copies of our system disks.

Step 23: Give yourself a pat on the back! You deserve it! You've created two newly configured system disks, and are now going to back them up to avoid potential disaster. You'll be relieved to know that Steps 1–22 need not be repeated to make backups. You *could* repeat them if you wanted the practice though. . . (fat chance, Geenen). You should still be in the Filer, by the way.

Take out the FORT2 backup disk you were supplied with when you purchased Apple FORTRAN. Place it in Drive 2, and place your newly configured FORT2 in Drive 1. Press **"R"**, then remove all files *except* SYSTEM.COMPILER from the backup FORT2. A sample run-through follows (note the somewhat strange response to the "REMOVE?" prompt):

```
REMOVE ? #5:?

REMOVE SYSTEM.COMPILER ? N
REMOVE SYSTEM.LIBRARY ? Y
REMOVE SYSTEM.APPLE ? Y
REMOVE SYSTEM.PASCAL ? Y
REMOVE SYSTEM.MISCINFO ? Y
REMOVE SYSTEM.CHARSET ? Y

UPDATE DIRECTORY ? Y
```

Remember that you may type **"N"** when asked whether or not to "UPDATE DIRECTORY?" if you've made a mistake.

Step 24: Now type **"T"** and answer the following prompts as shown. Note that you are transferring all files *except* SYSTEM.COMPILER, which is copy-protected. You probably will be warned that Units 4 and 5 have the same name; disregard the warning:

```
TRANSFER ? #4:?
TO WHERE ? #5:$

TRANSFER SYSTEM.COMPILER ? N
TRANSFER SYSTEM.EDITOR ? Y
TRANSFER SYSTEM.FILER ? Y
TRANSFER SYSTEM.LINKER ? Y
TRANSFER FORMATTER.CODE ? Y
TRANSFER FORMATTER.DATA ? Y
```

If you receive an "I/O error #16" message, remove the tape from the write-protected FORT2 disk, then start step 24 over. Be sure to replace the tape when finished.

You may now remove the two FORT2 disks; the backup is complete! Place a piece of tape over the backup FORT2's write-protect opening, as you did with the original.

Step 25: You will soon learn that disks used with Apple FORTRAN are formatted (read "initialized") differently from those used with Applesoft BASIC. Therefore, this step will be FORTRAN's equivalent to BASIC's "INIT HELLO".

NOTE: This process was not necessary for disk FORT2 because the backup you were given had already been formatted.

Place disk APPLE1 in Drive 1, and disk APPLE3 in Drive 2. Type **"Q"** to quit the Filer. Now press **"X"** and answer the prompt as follows:

```
EXECUTE WHAT FILE ? APPLE3:FORMATTER
```

After about 5 seconds, you are welcomed to the Formatter program, and asked which disk to format. Before responding, remove disk APPLE3 from Drive 2 and replace it with a blank diskette (this will become your FORT1 backup). After doing so, press **"5"** and then **"RETURN"**:

```
FORMAT WHICH DISK (4, 5, 9..12) ? 5
```

If you receive a message such as "DESTROY DIRECTORY OF FORT1:?", you probably have the wrong disk in Drive 2 (or your blank disk wasn't really blank). Type "N" and be sure to put a blank diskette in Drive 2.

The formatting process will take about 45 seconds. Upon its completion, you will again be asked which disk to format, but this time press **"RETURN"** to exit the program.

Step 26: Our Final Step! This backup step will be quicker than it was for FORT2, for now we simply transfer the *entire* disk. To do so, press **"F"** to enter the Filer. Remove disk APPLE1 from Drive 1, and replace it with your newly configured disk FORT1 (your newly formatted disk should still be in Drive 2; it happens to have the name BLANK). Type **"T"** to transfer files, and respond:

```
TRANSFER ? #4:
TO WHERE ? #5:

TRANSFER 280 BLOCKS (Y/N) ? Y

DESTROY BLANK: ? Y
```

After about one minute, the process will be complete. Place your FORT1 and FORT2 backup disks somewhere safe. Remember, they're your only recourse in the event of a system disk crash! FORT2 is especially valuable, since the SYSTEM.COMPILER file cannot be copied.

Step 27: Join me for a soda. It's on the house, as long as you promise not to spill it on FORT1 or FORT2 (ouch!).

G. WHY STUDY FORTRAN?

FORTRAN is the original high-level programming language. It was defined in April, 1957, and was standardized in 1966 by the American National Standards Institute, or ANSI.

Although it is an old language, it has been updated (the current ANSI standard was developed in 1977) and is still widely used. It is possible to write a very readable, well-structured FORTRAN program. This book will help you do so.

FORTRAN is one of the best programming languages available for producing output in tabular form. It is the language of choice for mathematicians, scientists, and engineers.

FORTRAN is an extremely fast language, since its executable programs are actually machine language programs. (Technical aside: actually, Apple FORTRAN generates what is called Pascal pseudo-machine code, or P-CODE, which is then run through an interpreter to convert it to Apple 6502 machine language. P-CODE is such a low-level language that for all practical purposes it *is* machine language. It should be noted that Apple FORTRAN's operating system and output code are based on the University of California–San Diego's Pascal P-System.)

Virtually all computer language programs (except those written in BASIC) are developed in the same manner as FORTRAN programs.

1. Major Differences from BASIC

a. FORTRAN requires an 80-character program line.
b. Line numbers are not required for the majority of program lines.
c. The positioning of commands in a program line is crucial.
d. Immediate execution of a newly written program is not possible.

2. Steps Required When Developing a FORTRAN Program

a. Create the program using a text editor.
b. Compile the text version of the program; the Compiler translates the entire FORTRAN program into machine language.
c. Link the machine language program with the necessary library routines.
d. Execute the linked program.

H. OPERATING SYSTEM OVERVIEW

IMPORTANT NOTE: Sections E and F of this introduction describe how to configure your FORTRAN system disks in a manner that most efficiently uses their very limited available disk space. All operating system discussion in this book will be based on this configuration. If your disks are set up as described in the FORTRAN Language Reference Manual, *you must reconfigure them to use this book!*

We begin our discussion of Apple FORTRAN by surveying its operating system.

1. Required Diskettes

a. FORT1 is the boot diskette. It contains the operating system's Command Level (a main menu, if you will), the System Library, and the workfile (your developing program).
b. FORT2 contains the Compiler, Editor, Filer, and Linker.
c. Your own *FORTRAN-Initialized* disk. One cannot save FORTRAN programs on a disk initialized in BASIC. Unfortunately, a disk may contain either BASIC or FORTRAN programs, but *not both.*

A disk may be formatted for FORTRAN by using the utility program FORMATTER.CODE on disk FORT2. Note that the Pascal and FORTRAN formats are interchangeable.

Once FORTRAN has been booted (see below), simply type **"X"** (for Xecute), and respond to the "EXECUTE WHAT FILE?" prompt with **"FORT2:FORMATTER"** (do *not* put quotes around your answer!). Once the program has been read in, you will see a new prompt: "FORMAT WHICH DISK (4, 5, 9...12)?". For reasons we will discuss later, Drive 1 is known as Device 4, while Drive 2 is Device 5. So place the disk to be formatted in either drive and type the appropriate response.

It will take about 45 seconds for the formatting process to be completed. Your disk will be given the name "BLANK", which you will be free to change later. To exit the formatter program, make sure disk FORT1 is back in Drive 1, then simply **press the RETURN key** without specifying a device number.

2. Booting FORTRAN

a. Place disk FORT1 (label-side up) in Drive 1. Close the disk drive's door.

b. Place disk FORT2 in Drive 2. Close its door, also.

c. Turn on the computer's monitor device.

d. Switch the computer on. (The switch is in the left rear of the machine.) If the machine is already on, you may use CTL-RESET and/or **"PR#6"** to achieve the same result.

e. After about 15 seconds, the screen will present you with a set of options called "Command Level." This is the outermost command branch of the operating system (each Command Level option contains its own set of sub-commands). Here is what appears on the screen:

```
COMMAND: E(DIT, R(UN, F(ILE, C(OMP, L(INK, X(ECUTE, A(SSEM, D(EBUG ? [1.1]
```

IMPORTANT NOTE: If you see a "[1.0]" at the end of the Command Level line, it means you have an older version of the operating system than this book will be describing. This older version differs in a few very minor ways from version 1.1. If your Command Level line contains a "[1.2]", you have the newest available version of the Apple Pascal operating system (it was released in late 1983). This version also differs very slightly from version 1.1. These differences will be mentioned when necessary. Fortunately, there is at least a 99% overlap between the three incarnations of the operating system.

3. Command Level Options

a. EDIT

Used to create and alter all FORTRAN program files.

b. RUN

Automatically compiles, links, and executes the workfile; this is a useful shortcut feature.

c. FILER

A file utility program; the Filer is used to get program listings, transfer files, list disk directories, and do other disk "housekeeping" chores.

d. COMPILE

Translates FORTRAN files into machine language.

e. LINK

Links machine language files with the necessary System Library routines.

f. XECUTE

Executes the specified file (but *only if the file has already been compiled and linked.*).

g. ASSEMBLE

Translates assembly language files into machine language (we will *not* learn assembly language in this book).

h. DEBUG

Reserved for future implementation; for now, this command is equivalent to "COMPILE". This option does not appear in operating system version 1.2.

4. Getting the Full Picture (Apple II Plus Users Only)

NOTE: FORTRAN programs require 80-character lines, but older Apples can handle only 40 characters per line. Therefore, these machines can display only HALF of the full screen at any one time. The following commands relate to this dilemma.

a. CTL-A

CTL-A is a toggle (switching back and forth) command. It selects either the left half or the right half of the screen. After booting FORTRAN, you must use CTL-A to see the full Command Level options list.

b. CTL-Z

It can quickly become annoying to press CTL-A repeatedly. Pressing CTL-Z will initiate "Auto-Follow" mode; now the screen will literally follow the cursor along the program line (with no need for you to manually select left or right screen). CTL-Z is another toggle command, so simply press CTL-Z again to disengage it.

c. Lower-case display

Although the Editor allows Apple II Plus users to simulate lower-case

display (and actually have it generate lower case when information is sent to a printer, or when the Editor is used on a machine with true lower-case display), a relatively inexpensive lower-case chip would be a wise investment. With this chip, mixed-case text (an important element of a readable program) will actually appear on the monitor.

In the absence of a lower-case chip, CTL-E will serve as a CAPS LOCK key (it also happens to be a toggle, like CTL-A and CTL-Z). When you are using CTL-E, all upper-case text will appear in inverse, while lower-case text will appear normal.

5. Automatic 80-Column Display (Apple IIe and IIc Users Only)

When Apple FORTRAN is booted on an APPLE IIe or IIc equipped with the 80-column text card, display will automatically come up in 80 columns and mixed case.

NOTE: THE OPERATING SYSTEM ACCEPTS BOTH UPPER- AND LOWER-CASE LETTERS, BUT DOES NOT DISTINGUISH BETWEEN THE TWO WHEN INTERPRETING KEYWORDS (lower-case letters are automatically converted to upper-case during compilation). Lower-case letters in character strings are left intact.

I. YOUR FIRST FORTRAN PROGRAM

Your first FORTRAN program's purpose will be to generate a simple table for figuring a car's gas mileage. For simplicity, we will assume that your car has a 15-gallon tank. A loop changes the number of miles traveled on one tank from 300 to 600 in increments of 50 miles. It will look like this once it's finished:

```
        PROGRAM GASUSE

        WRITE (*, 10) '>>> Computes gas mileage <<<'
10      FORMAT (/, A, /)
        WRITE (*, 20) 'Miles', 'Gallons', 'MPG'
20      FORMAT (A, 5X, A, 5X, A)
        GALLNS = 15.0

        DO 40 MILES = 300, 600, 50
           WRITE (*, 30) MILES, GALLNS, MILES/GALLNS
30         FORMAT (1X, I3, 8X, F4.1, 6X, F4.1)
40      CONTINUE

        END
```

Do not worry about memorizing nor fully understanding all the steps we will be going through. These procedures will be presented in full detail later. The function of this exercise is merely to help you overcome "opening-day jitters" by giving you a feel for how FORTRAN programs are developed and executed.

Let's go for it!

1. Follow the normal start-up procedure (see letter H, Section 2, "BOOTING FORTRAN") to get FORTRAN up and running.

2. You are now at Command Level. Those using machines with 40-column output should now use CTL-A to become accustomed to flipping to right- and left-screen displays.

3. Choose Command Level option **"E"** to enter the Editor (Do this by pressing the letter "E" on the keyboard; you need not press RETURN). Note that Drive 2's "IN USE" light comes on when you do so, indicating the presence of the Editor program on disk FORT2.

4. You should now see this prompt:

```
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE, <ESC-RET> TO EXIT) :
```

The above prompt **SHOULD** read "<RET> for NEW file", for you now **press the RETURN key** to inform the Editor that you will be creating a new file. Again notice that FORT2 spins when you press RETURN. This is because an overlay of the large Editor program must be read in.

NOTE: IF THE ABOVE PROMPT DID NOT SHOW UP, but rather a file was read into the Editor, it means that the previous user forgot to do a little housekeeping. To recover from this person's mistake, press the following keys (do NOT use any spaces and do NOT press the RETURN key):

```
Q E F N Y Q E
```

translated, this means:

```
Q(UIT THE EDITOR
E(XIT WITHOUT UPDATING THE WORKFILE
F(ILER
N(EW WORKFILE
Y(ES, THROW AWAY OLD WORKFILE
Q(UIT THE FILER
E(DITOR
```

Now you should see the "NO WORKFILE IS PRESENT" prompt.

5. You should now see the Editor's prompt line. (From now on, it will be taken for granted that those using 40-column Apples will need to use CTL-A to see the full 80-column display.) It looks like this:

```
>EDIT: A(DJST C(PY D(LETE F(IND I(NSRT J(MP R(PLACE Q(UIT X(CHNG Z(AP
```

6. Press the **"I"** key to choose option "I(NSRT". Notice that the prompt line on the top of the screen changes.

7. You will soon learn that FORTRAN instructions must begin no earlier than Column 7 (only line numbers may begin in Column 1). Using the TAB key (or its equivalent on the Apple II Plus, CTL-I), will save you from having to press the spacebar six times; the TAB key will actually start you out in Column 9.

Press TAB or CTL-I, then type the first program line:

```
PROGRAM GASUSE
```

Even though GASUSE is really two words, do *not* use any spaces!

NOTE: IF AT ANY TIME YOU MAKE A MISTAKE WHEN TYPING, USE THE LEFT-ARROW KEY TO MOVE BACK TO THE ERROR (EVEN IF IT'S ON A DIFFERENT LINE). You will notice that moving the cursor backwards ERASES all text it passes over.

8. Now **press the RETURN key TWICE.** Note that this leaves a blank line below the program name. Blank lines, which serve to make a program much easier for you and me to read, will be ignored by the FORTRAN compiler.

Also note that the Editor will automatically indent each new line to the indentation level of the previous line. This saves us from having to constantly press the TAB key.

9. Enter the second line of the program. This is a good time for those using Apple IIe's or IIc's to become familiar with clicking the "CAPS LOCK" key in and out. If you are using an Apple II Plus without a lower-case chip, use CTL-E as your shift-lock key; note that inverse characters represent capital letters:

```
WRITE (*, 10) '>>> Computes gas mileage <<<'
```

Except for the text enclosed by the single quotes, spaces in this line are *not* significant (if we assume the line begins no earlier than Column 7). The following would have been equally correct:

```
WRITE(*,10) '>>> Computes gas mileage <<<'
WRITE     (*,10)        '>>> Computes gas mileage <<<'
write (*, 10) '>>> Computes gas mileage <<<'
```

Note that even lower case is acceptable for the keyword "WRITE". As a rule, we will always capitalize keywords, use mixed case for messages and comment lines, and try to remain consistent with an easy-to-read spacing and indentation format.

10. Press RETURN. Our new line requires a line number over in Column 1, but the auto-indent feature has us in Column 9. No problem, just press the left-arrow key eight times. (If you press it more than eight times, don't panic! Just press RETURN one more time, and try it again.)

11. Now type:

```
10      FORMAT (/, A, /)
```

Don't forget to use the TAB key after typing in the line number. This way all of your instruction lines will line up nicely. From now on, it is assumed that you will be using RETURN to end all of your lines. Therefore, **press RETURN** if you haven't already done so.

12. Type the next three program lines, using the TAB and left-arrow keys as necessary to achieve proper indentation:

```
        WRITE (*, 20) 'Miles', 'Gallons', 'MPG'

20      FORMAT (A, 5X, A, 5X, A)

        GALLNS = 15.0
```

You have surely noted the "typo" GALLNS in the last line. Be assured that it *is* spelled correctly, for it is a variable. FORTRAN limits variables to a length of six characters, so a letter had to be dropped!

13. Place a blank line after the "GALLNS = 15.0" line (**press RETURN twice** instead of the usual once, after the line has been typed).

14. Now enter this line:

```
DO 40 MILES = 300, 600, 50
```

15. Notice in the complete program list that the next two lines are indented. These lines are the body of a loop. Your cursor should, at this point, be immediately beneath the "D" of "DO 40. . ." If it is not, **press the RETURN key**. Before typing the next line, **press the spacebar three times:**

```
   WRITE (*, 30) MILES, GALLNS, MILES/GALLNS
```

Again, don't "misspell" GALLNS.

16. The next line has a line number, so we must get back to Column 1. (The cursor should be in Column 12 after you close off the above line with RETURN.) **Press the left-arrow key 11 times** to do so. Too many left-arrow presses will cause you to wind up on the previous line. If this happens, press RETURN one more time, and try your luck at the left-arrow key again. The finished line should read:

```
30          FORMAT (1X, I3, 8X, F4.1, 6X, F4.1)
```

The first part of the section in parentheses reads "1 [the number "one"] X, I [the ninth letter, "I"] 3". Don't get these two mixed up!

17. The second-to-last typed line is:

```
40       CONTINUE
```

"CONTINUE" should again be in Column 9 (the same level as "DO" from an earlier line).

18. Finally, after **inserting a blank line,** place the word **"END"** as your last program line. *It is crucial that you press* RETURN *exactly once after the* END *statement, no more and no less! Be sure before going on that the cursor is directly below the* "E" *of* END.

19. You've finished writing the program! Your program should now look exactly like the program list given at the start of this exercise. **Type CTL-C.** (Do this by holding down the "CONTROL" or "CTRL" button, and pressing the "C" key *at the same time.*) This command makes your insertion permanent, and returns you to the Editor's prompt line (recall that you had earlier chosen the "I(NSRT" option).

20. Now we're ready to quit the Editor, save the program on disk, and translate it to machine language so it may be executed. *Without using spaces or pressing RETURN,* type these keys:

```
Q U R
```

They mean:

```
Q(UIT THE EDITOR
U(PDATE THE WORKFILE
R(UN THE PROGRAM
```

When you type the "Q", you will see Drive 2 kick in; another of the Editor's overlays is now being summoned. Typing the "U" causes Drive 1 to finally stir. That is because your "workfile" (or program) is being saved on FORT1.

Finally, you will notice that the "R" sets off a chain reaction of events. The console will inform you that you are now in the process of compiling. After about 15 seconds, you will see the prompt:

```
LISTING FILE?
```

All you need do is **press the RETURN key.** Now you witness some rather cryptic messages, and see a series of dots printed out on the monitor (more on these later).

Approximately 20 seconds later, you will be informed that the operating system is linking. Again you will see some strange prose like "LINKING MAIN-SEGX".

NOTE: If you weren't careful typing the original program, you may have been stopped somewhere during the compiling stage with an error message. If so, type the letter "E" and you will be brought back to the Editor; your program will be read in automatically. At this point, I recommend that you follow the instructions listed in Step 4 for getting rid of an old file (Q E F N Y Q E); you may then start over. Be more careful this time; you're beginning to see how temperamental the Compiler can be.

Finally, after about 30 seconds' worth of linking (and about 60 seconds after pressing "R"), you will see your program being executed.

21. Now that you've successfully executed your first FORTRAN program, let's see if we can change it a bit. Type the letter **"E".** The Editor will come up and read in your program for you. Once it has, **use CTL-L** to move down to the "GALLNS = 15.Ø" line.

NOTE: The up- and down-arrow keys of the Apple IIe or IIc will not be functional until you run the utility setup program described in Chapter 1, Topic D, "Cursor Movement." If CTL-L does not work, try the down-arrow key. If it works, the setup program has already been executed.

Use the right-arrow key to move the cursor so it is on the "5" of "15.0". Note that your program is unaltered by these cursor moves.

Choose option "X)CHANGE" by typing an **"X".** Now type **"7.5"** (*without* the quotes). Your line should now read "GALLNS = 17.5". **Type CTL-C** to make your exchange permanent; the Editor prompt line should now return at the top of the screen.

Now use **CTL-L** or down-arrow to move down two more lines to "DO 40 MILES = 300, 600, 50". Your cursor should be between the "=" and "300".

Choose the "**R**(EPLACE" option, and respond to the new prompt with:

```
/50//25/
```

Notice that you didn't need to press RETURN. Did you see what happened? Your line should now read "DO 40 MILES = 300, 600, 25".

Now we'll go through the stages to get the program executed once more. Type these four keys in rapid succession *(note that <RETURN> means "press the* RETURN *key"):*

```
Q U R <RETURN>
```

which once again means:

```
Q(UIT THE EDITOR

U(PDATE THE WORKFILE

R(UN THE PROGRAM

<RETURN> is the answer to the Compiler's "LISTING FILE ?" prompt
```

The operating system has a buffer which can store answers to prompts even before the prompts appear—*unless you have the older version 1.0, which does NOT have a buffer; in this case, you must type the keys only AFTER the prompts appear.* You will, I hope, see a slightly different version of the first program appear after passing about 60 pleasant seconds with the Compiler and the Linker. Can you infer how the changes we made affected the program's output?

22. If you haven't already had the pleasure, now is a good time to become familiar with the Compiler's method of reporting errors. From Command

Level, choose once again the "E(DIT" option. Your program should be read in for you. Choose the Editor's "**R**(EPLACE" option. Respond to the prompt with:

```
/GALLNS//GALLONS/
```

Now press the "Fab Four" once again to get rolling:

```
Q U R <RETURN>
```

Notice that when the error is reported (with a beep), you may press ESC to exit the Compiler, spacebar to check for further errors, or "E" to return to the Editor. Choose **"E".** After your program is read into the Editor, note how the error message remains on the screen until you press the spacebar. Also notice that you are positioned exactly two lines below the erroneous line. This will always be true; it's too bad you aren't placed right *on* the line at fault.

You may recover from the error by **pressing the spacebar** (removing the error message and bringing the Editor prompt line up), then **using CTL-O** or up-arrow, if the setup has already made it functional, to move up two lines (note that this is the *letter* "O," not the *digit* "Ø"), and finally **the right-arrow key** to move to the "N" of "GALLONS".

Now select the "**D**(ELETE" option, and **press the left-arrow key once.** Pressing **CTL-C** affirms the deletion, and will allow you to try out the corrected file, if you wish (you won't be told how to get it running this time!).

23. Once you've finished experimenting, be sure to remove your file so the next person using FORTRAN won't have your file read in when summoning the Editor (assuming you're sharing disks in a classroom). To do so, type:

```
F N Y Q
```

that is:

```
F(ILER

N(EW WORKFILE

Y(ES, THROW AWAY OLD WORKFILE

Q(UIT FILER
```

Command Level should now reappear, and the computer is ready for the next person to use.

J. YOUR SECOND FORTRAN PROGRAM

This time, I'm going to be a little bit more stingy with directions, for you should have picked up some "know-how" by going through the first program.

The purpose of the second program is to simulate the sound of a siren, and inform a person that he has been arrested. The siren sound is generated with three loops. An outer loop controls the number of siren cycles we will go through—in this case, 15. There are two loops nested within the outer loop. The first controls the ascending portion of the siren cycle which ranges from pitches of 30 up to 50, while the second takes care of the descending phase with pitches of 50 down to 30. It sounds so real, you may be able to fool someone! Here is the completed version:

```
$       USES APPLESTUFF
        PROGRAM SIREN
        INTEGER CYCLES, PITCH

        DO 30 CYCLES = 1, 15

           DO 10 PITCH = 30, 50
              CALL NOTE (PITCH, 1)
10         CONTINUE

           DO 20 PITCH = 50, 30, -1
              CALL NOTE (PITCH, 1)
20         CONTINUE

30      CONTINUE

        WRITE (*, 40)
     *     CHAR(15), 'You''re under arrest, Joe Fortran!', CHAR(14)
40      FORMAT (A, A, A)

        END
```

Here we go!

1. Boot FORTRAN.

2. Type **"E"** to enter the Editor.

3. Press **"RETURN"** when you see the prompt "NO WORKFILE IS PRESENT. . . , etc.". If this prompt doesn't appear, recover by typing these 7 keys: **"Q E F N Y Q E".** Recall that this key sequence serves to remove a previous workfile.

4. Choose Editor prompt line option "**I**(nsert".

5. Type a "**$**" in Column 1 of the first line, then **press TAB or CTL-I,** and then type "USES APPLESTUFF". **Press RETURN** to get to the second line. *Recall that you may use the left-arrow key to back up to any mistakes you may have made, even if they happen to be on a different line.*

6. Press TAB, then type "PROGRAM SIREN". Continue with the third line "INTEGER CYCLES, PITCH". Did you remember the Editor's auto-indent feature? Recall that this saves you from repeatedly pressing TAB at the start of each line.

7. Press RETURN *TWICE* to generate a blank line, then type "DO 30 CYCLES = 1, 15".

8. After leaving another blank line, **press the spacebar 3 times,** followed by "DO 10 PITCH = 30, 50".

9. The next line, "CALL NOTE (PITCH, 1)", should be **preceded by three spaces.** (Hopefully you're looking at the completed version of the program to check the indentation we're using.)

10. Note that the next line needs a line number, so we need to backtrack to Column 1. **Press the left-arrow key 14 times,** type **"10",** then **TAB,** then **three spaces,** and finally **"CONTINUE".**

11. Insert a blank line, then **press TAB, 3 spaces,** and "DO 20 PITCH = 50, 30, -1".

12. You're on your own for the next four typed lines ("CALL NOTE (PITCH, 1)", "20 CONTINUE", "30 CONTINUE", and "WRITE (*, 40)"). Watch the indentation, and don't forget to add the blank lines for increased readability.

13. The cursor should now be below the "W" of "WRITE (*, 40)". The next line is actually part of the previous line! In FORTRAN, an "*" in Column 6 means that the forthcoming line is to be considered an extension (or wrap-around) of the previous line.

Press the left-arrow key three times to get back to Column 6, then type "*", **press TAB,** followed by **three spaces,** and then the rest of the line: "CHAR(15), 'You"re under arrest, Joe FORTRAN!', CHAR(14)".

Note the TWO (side-by-side) single quotes in the word "You"re". This is necessary since FORTRAN uses the single quote for two purposes: to begin and end messages, and as an apostrophe. Two ADJACENT single quotes will appear

as ONE apostrophe when the program is run. Also note that ONE DOUBLE *quote character CANNOT be substituted for TWO* SINGLE *quote characters.*

"CHAR(15)" generates inverse printout on an Apple IIe or IIc (it has no effect on an Apple II Plus), while "CHAR(14)" returns printout to normal.

14. You should be able to handle the rest of the program on your own. *Don't forget that you must press* RETURN *ONCE AND ONLY ONCE after* END.

15. Do you remember how to initiate compiling and execution? That's right, press the Fab Four: **"Q U R <RETURN>".**

If you receive an error message while compiling (shame on you!), you'll have to return to Step 2 and start all over. Be more careful this time!

If your program compiles and links, but you receive the message "STACK OVERFLOW" when the program is running, it means that your FORT1 disk does not contain the SYSTEM.STARTUP file as described earlier in letter F, "Arriving at the new configuration," Steps 17-22.

If your program runs correctly, you may proceed.

16. Pretty slick, huh? If you want to execute it again, simply press **"U"**, which stands for "User Restart". In English, this means re-execute the program that was last executed—this is a neat little trick to remember, by the way.

17. I hereby issue this challenge: Can you return to the Editor and change the number of siren cycles from 15 to 25 (or any other number), as well as change the name of the offender?

18. When you are finished, it is always your responsibility to get rid of the workfile. To do so, type **"F N Y Q"** from Command Level.

Part I
THE APPLE FORTRAN OPERATING SYSTEM

Chapter 1:
EDITOR

NOTE: THE EDITOR RESIDES ON DISKETTE FORT2, SO FORT2 MUST BE PRESENT IN ONE OF THE DRIVES (USUALLY DRIVE 2) BEFORE YOU CALL IT IN. The Editor program is much too large to fit in memory, so at any given time only a small portion (an OVERLAY) will be present in memory. THIS NECESSITATES LEAVING FORT2 IN DRIVE 2 AT ALL TIMES, so a different overlay may be called in if needed.

A. GETTING INTO THE EDITOR

To call in the text Editor, simply type **"E"** for E(dit from Command Level. You need not press RETURN after typing the letter for your Command Level option.

B. OPTIONS UPON CALLING IN THE EDITOR

1. Loading the Workfile

If you have a workfile (a scratchpad copy of your developing program, saved on FORT1 under the special file name SYSTEM.WRK.TEXT), it will be read in and displayed automatically by the Editor. We will cover the creation of a workfile later in this lesson.

2. Loading a Previously Created File

If a workfile is not present, you will see this prompt:

```
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE, <ESC-RET> TO EXIT )
```

If you now wish to edit a previously created file, type the name of the disk it is on, followed by a colon, then the file name itself. Following is an example (as usual, the user reply is in **emphasized print**):

```
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE, <ESC-RET> TO EXIT )
: FORTRAN:PROGRAM
```

The colon serves to separate the disk name from the file name. In this example, the disk name is "FORTRAN", the program file name is "PROGRAM.TEXT".

NOTE: At this time, two important points need explanation:

1. *Every FORTRAN disk has its own name. This name consists of at most 7 keyboard characters, NOT just letters and/or digits, and NOT necessarily with a letter first. A colon separates the disk name from the file name.*
2. *Every program file name contains a SUFFIX. The suffix serves to distinguish between a text file and a machine language file (more on this distinction later). A period separates the file name from its suffix. The file name and suffix together may be at most 15 keyboard characters. NOTE THAT IT IS NOT NECESSARY TO TYPE THE ".TEXT" SUFFIX WHEN ANSWERING THE ABOVE PROMPT. This is one of many shortcuts available to you if you use the suffixes ".TEXT" and ".CODE".*

To call in a file from a disk, one must have that disk present in one of the drives. Since the Editor is on FORT2 (which should be in Drive 2 at all times), *you must remove FORT1 from Drive 1 and replace it with the disk containing your program before calling it in.*

3. Creating a New File

You may also create an entirely *new* file. To do so, simply press **RETURN** when you're given the prompt:

```
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE, <ESC-RET> TO EXIT )
: <RETURN>
```

4. Exiting the Editor

In case you entered the Editor accidentally, you may press **ESC** followed by **RETURN** to go back to Command Level.

C. THE EDITOR PROMPT LINE

1. Adjust

Adjust is used mainly in word processing applications. Its submenu is:

```
ADJUST: L(JUST R(JUST C(ENTER <LEFT,RIGHT,UP,DOWN-ARROWS> [<ETX> TO LEAVE]
```

The only option we need concern ourselves with at the present time is Left Justify. If you forget to give a line a line number, the Editor will not let you put it in unless you first adjust the line from Column 9 to the extreme left margin (Column 1). To do so, first move the cursor to the line, then type **"A"** (for Adjust), **"L"** (for Left Justify), then **CTL-C** (to get out of Adjust mode; *"ETX" in this and other prompts refers to CTL-C*). Now insert your line number (see Insert, below).

You will learn more about Adjust in Chapter 20, as well as in one of the exercises at the end of the chapter.

2. Copy

Copy is a word processing command which we will discuss in Chapter 15.

3. Delete

The Delete prompt looks like this:

```
DELETE: < > <MOVING COMMANDS> [<ETX> TO DELETE, <ESC> TO ABORT]
```

In Delete mode, you use the **left- and/or right-arrow keys** to delete text one *character* at a time (hence the "< >" in the prompt line), or the **"RETURN"** key to delete one *line* at a time ("RETURN" is one of the cursor moving commands we will discuss in a few moments). After deleting text, you have two options:

1. **CTL-C:** pressing CTL-C will make the deletion *permanent.*
2. **ESC:** pressing the ESC key will cancel the deletion and restore the deleted text.

After pressing CTL-C or ESC, you will be returned to the Editor prompt line.

4. Find

Searches in the set direction (specified by either a ">" or a "<" in the prompt line) for the specified character string. The cursor will then be placed in front of that string. Here is the Find prompt line:

```
>FIND [1]: L(IT <TARGET> =>
```

">" means that the search will be *forward* from the cursor.

"[1]" means it will find the *first* occurrence of the specified string.

"L(IT" means that to initiate *literal* search, one must type an "L"; the default search is *token.*

DEFINITIONS:

LITERAL: finds the specified string *whether or not it is a complete word.*

TOKEN: finds the specified string *only if it is a word, i.e., only if it is preceded and followed by a blank or a "RETURN".*

"<TARGET> =>" simply tells you to type in the string to find. A typical application is to find a misspelled word. Notice how the answer is printed on the same line as the prompt:

```
>FIND [1]: L(IT <TARGET> => /AFFRICA/
```

Also note that the string must be enclosed within slashes or "delimiters". Here is how to use literal search:

```
>FIND [1]: L(IT <TARGET> => L/AFFRICA/
```

Note that L (for literal) is typed *before* the delimiters and the string. The above *literal* search would find the typo "AFFRICAN", for example, whereas *token* search would not. This is because "AFFRICA" is only a *portion* of the word "AFFRICAN".

WARNING: "FIND" WILL NOT WRAP AROUND SHOULD IT COME UPON THE END OF THE FILE. If you're on the last line of your program and want to move the cursor to one of the first lines, you must reverse the direction of the arrow (see below), then answer the Find prompt.

5. Insert

The Insert prompt line follows:

```
INSERT: TEXT (<BS> A CHAR, <DEL> A LINE) [<ETX> ACCEPTS, <ESC> ESCAPES]
```

After you choose Insert, all characters typed will be inserted *to the immediate left of the cursor.* The rest of the file (if it exists) is moved over to make room for the insertion.

NOTE: It is possible to insert a blank line in a program by pressing the RETURN key twice at the end of a line.

Insertions must be terminated by either **CTL-C** (making it permanent) or **ESC** (canceling the insertion), whereupon you will be returned to the Editor prompt line. By the way, the "BS" in the prompt line doesn't mean what you think it means; rather, it means "backspace a character." In other words, it's merely to inform you that you may use the left-arrow key to back up, one character at a time, to any mistakes you may have made.

NOTE: To begin a new program, you must choose I(NSERT before you type or else nothing will happen. Then, upon finishing, you must type CTL-C or else you may lose your entire program.

Those using an Apple II Plus without a lower-case chip may use **CTL-E** as a "CAPS LOCK" key when inserting. Normally displayed letters will appear as lower-case letters when printed on a printer with mixed-case capability, while inverse letters will appear as capitals.

6. Jump

You have four options for Jump. Type the first letter of your choice:

1. B(EGINNING: jumps cursor to the beginning of the file
2. E(ND: guess what this does?—Right!
3. M(ARKER: is discussed in Chapter 20
4. ESC: gets you out of Jump mode

7. Replace

Finds *and* replaces text. Here is the prompt:

```
>REPLACE [1]: L(IT V(FY <TARG> <SUB> =>
```

"**>**" is the direction of search.

"[1]" means it will find and replace the first occurrence.

"L(IT" means you must type an "L" to initiate literal search; the default search is token.

"V(FY" indicates that you must type a "V" to initiate Verify mode. Verify is a useful option which *will ask you whether or not it should replace the word—once it is found.* Otherwise, replacement is automatic.

"**<TARG> <SUB> =>**" indicates that you must first identify the word to be changed (TARGET), followed by its replacement (SUBSTITUTE).

With Replace, you may not only *find* typos, you may *correct* them. Here are some valid answers to the prompt:

```
>REPLACE [1]: L(IT V(FY <TARG> <SUB> => /AFFRICA//AFRICA/
```

```
>REPLACE [1]: L(IT V(FY <TARG> <SUB> => V/GENEN//GEENEN/
```

Notice that the "V" for "Verify option" in the second example is typed *before* the delimiters and strings. Here is what you will see when you find a target string with Verify engaged ("␣" represents pressing the spacebar):

```
'R' REPLACES '␣' DOESN'T <ESC> ABORTS
```

WARNING: "REPLACE", LIKE "FIND", DOES NOT WRAP AROUND WHEN SEARCHING. Be aware of the direction of your search!

8. Quit

Presents you with five choices:

a. U(PDATE THE WORKFILE AND LEAVE

This option saves the current file under the name SYSTEM.WRK.TEXT, thus creating your workfile. Recall that this file will be read in automatically when you invoke the Editor.

NOTE: Workfiles have several advantages in later stages of program development, too. The workfile is saved on disk FORT1, SO FORT1 MUST BE IN DRIVE 1 WHEN YOU CHOOSE THIS OPTION! After updating, you will be returned to Command Level.

b. E(XIT WITHOUT UPDATING

This is a dangerous option because it does not save anything! You will be returned to Command Level if you answer "Y" to the prompt "THROW AWAY CHANGES SINCE LAST UPDATE?". Fortunately, if you answer "N", you will be returned to the Editor with your file intact.

The main reason for this option is to allow a means of recovery when a previous user's workfile is read into the Editor. When this happens, the current user would like to get out of the Editor in order to call in the Filer (to issue its N(EW WORKFILE command).

c. R(ETURN TO THE EDITOR WITHOUT UPDATING

This option can be used if you forgot something and want to go back to the Editor.

d. W(RITE TO A FILE NAME AND RETURN

If you want to save the file under a name of your choice, or on a disk other than FORT1 (this disk must, of course, be present in Drive 1), use this option. After saving the file, you will receive the self-explanatory prompt "DO YOU WANT TO E(XIT OR R(ETURN TO THE EDITOR?". If you decide to exit, you will be returned to Command Level, so be sure FORT1 is back in Drive 1.

e. S(AVE WITH SAME NAME AND RETURN

This option saves the program under the same name by which it was read in, and it is invaluable for protecting yourself from disaster! By saving your program every ten minutes or so, you avoid the frustrating possibility of losing your entire program because of a power outage, someone tripping on the power cord, or some other mishap.

The only prompt you need to answer is this: "PURGE OLD BIG:BUCKS.TEXT?" or whatever name the program was given. This is just to remind you that you are replacing an old version of the program with a new version, so type "Y".

NOTE: This option does not appear in operating system version 1.0.

9. XCHANGE

If you want to trade one character or sequence of characters for another, use this option, which is similar to editing in BASIC. The characters you type will simply be written over the old characters. After exchanging, you must again press either **CTL-C** or **ESC.**

10. ZAP

Zap is of limited usefulness to novice users and will not be discussed. Curious users are hereby referred to the *Apple Pascal Operating System Reference Manual.*

11. VERIFY

This command will redisplay the current Editor text "page" with the cursor in the center of the screen. This allows one to see an equal amount of text both before and after the cursor in large programs. You may have noticed that this command does not appear in the Editor's prompt line.

D. CURSOR MOVEMENT

Note that almost all Editor commands are relative to cursor position. Here is how the cursor can be moved *without affecting the contents of the file.* The following commands may be typed any time the Editor prompt line is showing:

1. Right-Arrow Key

moves you to the right, one character at a time.

2. Left-Arrow Key

moves you to the left, one character at a time.

3. Up-Arrow Key (CTL-O on Apple II Plus)

moves you directly up one line.

NOTE: Apple IIe and IIc users must use the utility program SETUP.CODE *on Apple Pascal disk APPLE3 to make the up- and down-arrow keys functional (unless you have operating system 1.2; in this case, these keys are already operative). Fortunately, this rather intricate process need be executed only ONCE. Here is a brief rundown of how to do so:*

1. Place disk FORT1 in Drive 1, and disk APPLE3 in Drive 2.

2. Start the computer (using the power switch, or "PR#6").

3. From Command Level, type **"X"** (for X(ecute).

4. Respond to the "EXECUTE WHAT FILE?" prompt with "APPLE3:SETUP"—do *not* type the quotes. After being told that the computer is "initializing" for a few seconds, you will see a small menu.

5. Type **"C"** to implement the "C(HANGE" option. As you will see, this option has its own submenu.

6. Choose to make a **S(INGLE** change.

7. Respond to the "NAME OF FIELD:" prompt with "KEY TO MOVE CURSOR UP" (again, *don't* type the quotes). After pressing **RETURN** you will see the present value for this field (in several different representations: octal, hex, etc.); it should be listed as CTL-O (or ∧O in the program's notation).

You will now be asked "WANT TO CHANGE THIS VALUE?". Press the **"Y"** key.

8. Respond to the "NEW VALUE:" prompt by pressing the **up-arrow key.** The screen will display a "?" when you do so. Don't worry, the computer is not confused, it simply can't display an up-arrow character.

Press RETURN to make the change official. You will again see the list of representations; the value should now read CTL-K (∧K is what you will actually see), which is the character generated by the up-arrow key.

The computer will again ask "WANT TO CHANGE THIS VALUE?" just to make sure you've got it right. Unless something went wrong, type **"N".** You're now back to the CHANGE menu.

9. Again choose the S(INGLE change option.

10. This time, enter "KEY TO MOVE CURSOR DOWN" as your field to change. The value listed should be CTL-L (∧L). You then answer "WANT TO CHANGE THIS VALUE?" with **"Y".**

11. Press the **down-arrow key** in response to "NEW VALUE:". Again you'll see a "?" when you do so; as before, the computer can't display a down-arrow character.

Press RETURN, and you will see the value listed as CTL-J. You should now be able to type **"N"** when asked "WANT TO CHANGE THIS VALUE?".

12. Choose to Q(UIT the CHANGE option. You're returned now to the original SETUP menu.

13. Q(UIT the Setup program. You'll be given several options at this point.

14. Type a **"D"** to specify a D(ISK UPDATE. You should now see Drive 1 kick in. Your new setup has been saved as FORT1:NEW.MISCINFO.

15. E(XIT the Setup program. You are now back at Command Level.

16. Replace disk APPLE3 with FORT2.

17. We are now going to make file FORT1:NEW.MISCINFO into a permanent *system* disk file. To begin, enter the F(ILER.

18. Choose option C(HANGE by typing a **"C".**

19. Answer the "CHANGE?" prompt with **"NEW.MISCINFO"** (again, no quotes).

20. When asked "CHANGE TO WHAT?", reply **"SYSTEM.MISCINFO".**

21. Answer **"Y"** when asked "REMOVE OLD FORT1:SYSTEM.MISCINFO?". We've now replaced the old setup file with our new one.

22. The removal of the old file left a "hole" in disk FORT1. Since room is so precious on this disk, we are going to eliminate it. Type **"K"** (for K(RUNCH).

23. Respond to the "CRUNCH?" prompt with "FORT1:"; type no quotes, but *do* type the colon.

24. Answer **"Y"** when asked "FROM END OF DISK, BLOCK 280?". You will now be informed that several files are being moved forward.

25. **Q**(UIT the Filer.

26. Now that you're back at Command Level, choose to **I**(NITIALIZE. This causes the system to do a "warm boot," which in essence informs your Apple of the setup changes you've introduced. From now on, you may use the up- and down-arrow keys *without* going through these 26 steps. Your setup has become a permanent system file!

4. Down-Arrow Key (CTL-L on Apple II Plus)

moves you directly down one line.

5. Tab Key (CTL-I on Apple II Plus)

moves one tab stop (every eight characters) *in the set direction.*

6. Return

moves, again *in the set direction,* from the current cursor position to the *start* of the next line.

7. P

moves one page *in the set direction.* A *page* is equivalent to one screen width, or 24 lines. As the programs you write get longer and longer, it is often necessary to go to widely separated parts of the same program. Page allows you to do so either forward or backward one screen width at a time.

NOTE: In addition to being used to move the cursor, the movement keys may be used in Delete mode; for instance, TAB would delete to the next tab stop.

E. CHANGING DIRECTIONS

If you would like to change the direction the prompt-line arrow is facing, type one of the following keys any time the Editor prompt-line is showing:

1. > changes direction to forward

2. < changes direction to backward

F. REPEAT FACTORS

Most of you are probably aware that keys may be repeated. On an Apple IIe or IIc, simply hold down the key. On an Apple II Plus, hold down the key and the REPT button. The problem with this is that the Apple FORTRAN operating system has a buffer (unless you have version 1.0). This buffer can store keystrokes before they can even be displayed. When you hold down the right-arrow key, you may be generating, say, 15 characters per second, while the Editor can move the cursor perhaps only 10 characters per second. So what happens if you press the right-arrow key for four seconds? Your cursor will have moved 40 characters, but you have really generated 60 characters. When you release the key, *the cursor will continue to move until all the characters in the buffer have been exhausted.* This means that there is some guesswork involved in moving the cursor in this manner. Fortunately, there is a better way.

Repeat factors allow a *precise* number of repetitions of a cursor move or specified command. As an added bonus, they also are much faster than the method described in the previous paragraph. A command is repeated as many times as possible if the special character "/" is used; otherwise, a repeat factor is an integer. Here are some examples:

5<UP-ARROW> : Move directly up 5 lines.

6<RETURN> : Move to the beginning of the 6th line from the cursor.

3P : Move 3 pages in the set direction.

/R : Replace ALL occurrences of the specified string.

Repeat factors may be used with *any* cursor movement command, and *only* with FIND or REPLACE from the Editor prompt line.

NOTE: The repeat factor is typed before the desired command.

G. REVIEW QUESTIONS

1. What is an overlay? Why is it important to remember that the Editor program contains overlays?

2. How can one get out of the Editor if it was entered accidentally?

3. List the three components of an Apple FORTRAN operating system file name.

4. Give the exact key sequence needed for replacing all occurrences of "WISCONSIN" with "MINNESOTA". Assume you are at the beginning of the file and that the directional arrow is facing forward.

5. List five cursor-movement keys.

6. What is a workfile? How is a workfile created?

7. How does one recover if an old workfile is read in after one calls in the Editor?

H. EXERCISES

1. In this exercise, you will learn how to use the Adjust command, and gain some familiarity with the operating system's buffer.

a. Enter PROGRAM GASUSE from the previous section of the book, *but omit the line numbers, and type all the text in Column 1,* as follows:

```
PROGRAM GASUSE

WRITE (*, 10) '>>> Computes gas mileage <<<'
FORMAT (/, A, /)
WRITE (*, 20) 'Miles', 'Gallons', 'MPG'
FORMAT (A, 5X, A, 5X, A)
GALLNS = 15.0

DO 40 MILES = 300, 600, 50
WRITE (*, 30) MILES, GALLNS, MILES/GALLNS
FORMAT (1X, I3, 8X, F4.1, 6X, F4.1)
CONTINUE

END
```

Don't forget **CTL-C** when you're finished!

b. **J**(ump to the **B**(eginning of the file.

c. Enter **A**(djust mode.

d. Try **R**(just. See how it moves the entire line to the right margin?

e. Now do a **C**(enter justify.

f. Press the **down-arrow key** (or else **CTL-L**). Press it again. Can you infer what the up- and down-arrow keys do in Adjust mode? Continue by centering the rest of the program.

g. Let's see if we can restore the text to its original format. Type **L**(just to pull the bottom line over to the left margin.

h. Hold down the **up-arrow key (CTL-O)** to handle the other lines,

but only until the cursor is halfway through the file. Did you see how the cursor kept right on moving *even after you released the key?* What's going on here?

i. Enough foolishness, we've got to get this file in shape! If we don't want the Compiler to spit it back out, we'll have to move all of our program lines over to at least Column 7 (we'll move them to Column 9, since that is where TAB moves them). Your cursor should now be on the first line. Press the **right-arrow key** eight times. Notice how it moves the line over *one character at a time.*

j. Type **13** and then **down-arrow** to move the rest of the lines over to Column 9. Note the use of a repeat factor.

k. Type **CTL-C** to get out of Adjust. There are two lines that need to be indented three extra columns ("WRITE (*, 30). . ." and the line immediately below it). See if you can move the cursor there, then adjust these two lines without my help. Don't forget **CTL-C** when you're finished.

l. Good work! Now all we need are line numbers. Move the cursor to the line below "WRITE (*, 10). . .". Try backing up the cursor to Column 1. No soap, right? Move back to the line now, and adjust the line to the left margin. After getting out of Adjust mode (**CTL-C**), type **I**(nsert, followed by **"10"** (the line number we need), then **TAB,** and **CTL-C.** This is a useful trick to know! You're on your own for the other three line numbers.

m. **J**(ump to the **B**(eginning of the file. Let's see how fast your computer is! Hold down the **right-arrow key** (and, if applicable, the **REPT** button) for four or five seconds, then let go. It has been my experience that the Apple IIe moves the cursor about as fast as the computer generates the characters, while the Apple II Plus has a tendency to lag behind. In any case, it would be wise to find out how heavily your computer relies on its buffer. *Note that the adjusting process took longer than a simple cursor move, so BOTH models needed the buffer back in* Step h.

n. I'll leave the rest up to you. Maybe you want to experiment with Adjust, the buffer, repeat factors, etc. Perhaps you'd like to run the program through the Compiler. (You want to *quit?* Gasp!) It's your move.

2. To familiarize my students with the Editor, I type several files containing familiar text—the school song, the national anthem, and so forth—with loads of misspelled words, omitted phrases, extra punctuation, and so on. I then save these files on a disk which all FORTRAN students share.

The students "patch up" the text by using as many different Editor commands as they possibly can.

NOTE: Students should choose the E(XIT WITHOUT UPDATING option when they Q(UIT the Editor; this, of course, leaves the "garbage version" intact for other students to repair.

If you have no garbage files at your disposal (no pun intended), make up your own. This is a fun, yet very valuable exercise, for mastery of the Editor will make the creation of future programs much easier!

Chapter 2:
FILER

A. ENTERING THE FILER

The Filer resides on disk FORT2 but, unlike the Editor, *it is written without an overlay.* In other words, once the Filer is read in from FORT2 (by typing **"F"** from Command Level), it is *not* necessary to leave FORT2 in one of the drives; the entire Filer program is now contained in memory.

B. THE FILER PROMPT LINE

The Filer prompt line, unlike other Command Level option prompt lines, contains *only single letters, not complete words.* Here it is:

```
FILER: G, S, N, L, R, C, T, D, Q
```

And there's more! To see the other options, type a question mark (**?**) and you'll see:

```
FILER: W, B, E, K, M, P, V, X, Z
```

Those using version 1.2 of the operating system will see the following two prompt lines:

```
G(et S(ave W(hat N(ew L(dir R(em C(hng T(rans D(ate Q(uit
```

and

```
B(ad-blks E(xt-dir K(rnch M(ake P(refix V(ols X(amine Z(ero
```

Note that your options are literally "spelled out" for you. In addition, you will note that your Filer prompts will be slightly different than those discussed in this chapter. Specifically, the prompts for List Directory, Bad Blocks Scan, Extended Directory List, Krunch, Prefix, Xamine, and Zero will have ". . .OF WHAT VOL?" appended to them, while the prompts for

Get, Remove, Change, Transfer, and Make will be followed by ". . .WHAT FILE?".

Following are the Filer options in detail:

1. GET

This command names a file other than FORT1:SYSTEM.WRK.TEXT as your workfile.

NOTE: THE "GET" COMMAND IS NOT PRACTICAL WITH TWO DRIVES (for technical reasons beyond the scope of the current discussion). The only way to create a workfile with a two-drive setup is through the use of the Editor's "U(PDATE THE WORKFILE" option. If you choose to do this with a previously created file, you must first read that file into the Editor (how?), then Q(UIT the Editor, and finally U(PDATE THE WORKFILE.

2. SAVE

This will save the workfile SYSTEM.WRK.TEXT (and the machine language version SYSTEM.WRK.CODE, if it exists) under the name you specify. *You should NOT specify a suffix when using* SAVE! Here is the prompt and a valid reply (the reply is in **emphasized print,** as always):

```
SAVE AS ? SLICK:SNOWFALL
```

after which you will see the operating system print:

```
FORT1:SYSTEM.WRK.TEXT
      --> SLICK:SNOWFALL.TEXT

FORT1:SYSTEM.WRK.CODE
      --> SLICK:SNOWFALL.CODE
```

Recall that the workfile is on disk FORT1, which is usually in Drive 1, while Drive 2 in this case should contain disk SLICK.

3. NEW

This command wipes out the workfile. You *must* do this before turning off the computer, or else the next person to use the computer will have *your* workfile automatically read in to the Editor (or, if you're not sharing disks in a classroom situation, your own *old* workfile will be read in). This option will ask you if you're sure you want to delete the workfile:

```
THROW AWAY CURRENT WORKFILE?
```

Unless you've changed your mind, type **"Y"**.

4. LIST DIRECTORY

NOTE: FOR YOU TO LIST A DISK'S DIRECTORY, THAT DISK MUST BE PRESENT IN ONE OF THE DRIVES.

The prompt you will receive is:

```
DIR LISTING OF ?
```

whereupon you respond with the name of any *disk (not program!) followed by a colon.* Example:

```
DIR LISTING OF ? DRASTIC:
```

You will now receive a catalog of the programs the disk contains, including the file length and date created for each file:

```
DRASTIC:
CHANGE.TEXT          6      3-FEB-84
CHANGE.CODE         29      3-FEB-84
FORMAT.TEXT         1Ø     12-MAR-84
FORMAT.CODE         32     12-MAR-84
4/4 FILES, 2Ø3 UNUSED, 185 IN LARGEST
```

The last line means that four out of four files are listed, that 2Ø3 out of 28Ø blocks are open for saving information, and that the largest single program you could save is 185 blocks long. Those using operating system 1.2 will also be notified of the number of *used* blocks on their disks.

Technical aside: A *block* contains 512 bytes of information; therefore, it is equivalent to two *sectors,* the units of file length used by Applesoft BASIC.

5. REMOVE

This command will delete the specified file. When answering the Remove prompt, *you must type a COMPLETE filename, INCLUDING THE DISK NAME AND SUFFIX.* Here's the prompt, along with a valid response:

```
REMOVE ? DRASTIC:CHANGE.TEXT
```

This means delete file CHANGE.TEXT from diskette DRASTIC. Then you'll see this rather strange message appear:

```
DRASTIC:CHANGE.TEXT --> REMOVED

UPDATE DIRECTORY ?
```

This means that you have one more chance to change your mind; type **"Y"** to make removal final, **"N"** to cancel deletion.

NOTE: If you reply to the Remove prompt with a file name only (no disk name), the filer assumes the program specified is on the boot diskette—FORT1.
Example:

```
REMOVE ? CHANGE.TEXT

FORT1:CHANGE.TEXT --> REMOVED

UPDATE DIRECTORY ? Y
```

There are two special shorthand characters in Remove. The "=" character is a *wildcard* specifier. Here is an example of its use:

```
REMOVE ? DRASTIC:CH=

DRASTIC:CHANGE.TEXT --> REMOVED
DRASTIC:CHANGE.CODE --> REMOVED

UPDATE DIRECTORY ? Y
```

This means remove *all* files beginning with "CH". This can be a dangerous device if you're not careful; luckily, the "UPDATE DIRECTORY ?" confirmation can save you from an unwanted deletion. To remove all files with suffix ".TEXT" from disk DRASTIC, you could respond "DRASTIC:=.TEXT" when given the "REMOVE?" prompt.

The other shorthand character is "?". Here is how it works:

```
REMOVE ? DRASTIC:?
```

after which you will see:

```
REMOVE DRASTIC:CHANGE.TEXT? Y
REMOVE DRASTIC:CHANGE.CODE? N
REMOVE DRASTIC:FORMAT.TEXT? N
REMOVE DRASTIC:FORMAT.CODE? Y

UPDATE DIRECTORY ? Y
```

NOTE: You are asked whether or not each file on the specified disk should be removed. All you have to do is answer either "Y" or "N".

6. Change

Allows you to change either a file name or a disk name. The prompts and replies:

```
CHANGE ? BLANK:DUMB.TEXT

CHANGE TO WHAT ? BLANK:SMART.TEXT
```

This changes only the name, *not the contents* of the file. Another:

```
CHANGE ? BLANK:

CHANGE TO WHAT ? ZELMO:
```

This changes the name of the disk from BLANK to ZELMO (*note the colons*). "BLANK:" is the name given to each disk during the initialization process; Change allows you to give your disk a more personalized name (remember the 7-character limit).

7. TRANSFER

This transfers files from disk to disk, from disk to printer, or from disk to monitor (TV screen). Prompts and replies (*note again that replies to prompts are COMPLETE filenames*):

```
TRANSFER ? ZELMO:SMART.TEXT

TO WHERE ? FORT1:$
```

The "$" means "gives it the same name on FORT1 as it has on ZELMO". The "=" and "?" shorthand characters also apply to Transfer. **"CP2:CH="** would mean "transfer all files beginning with CH from disk CP2"; **"CP2:?"** would list each file on disk CP2 and ask if it should be transferred.

```
TRANSFER ? ENGLISH.TEXT

TO WHERE ? PRINTER:
```

The above yields a printout of file ENGLISH.TEXT from diskette FORT1 (why FORT1?). Note the colon after PRINTER.

NOTES:

1. *Only* TEXT *(human-readable) files may be transferred to the printer or console. If you try to get a* CODE *file listing, the*

operating system will spit out garbage for about ten seconds, after which the system will reboot!

2. *If output from the Apple FORTRAN operating system is double-spaced on your printer, check its manual to see if you may disengage the printer's automatic line-feed feature (the operating system sends a line feed of its own). If this is not possible, you will have to execute program "LINEFEED.CODE" on Apple Pascal disk "APPLE3:" before a printout to achieve single spacing.*

```
TRANSFER ? BLANK:STARE.TEXT

TO WHERE ? CONSOLE:
```

Sends the file contents (again, TEXT files only) to the monitor. Note the colon after CONSOLE. You may use **CTL-S** to stop the program from scrolling off the screen too fast. To re-engage stalled output, press CTL-S again.

8. DATE

This command records the date on disk FORT1:

```
TODAY IS 14-JUN-84

NEW DATE ? 15

THE DATE IS 15-JUN-84
```

NOTE: If only the day is changed (and not the month) you need only type the day.

Another example:

```
TODAY IS 31-MAY-84

NEW DATE ? 1-JUN

THE DATE IS 1-JUN-84
```

9. QUIT

QUIT leaves the Filer and returns to Command Level, so *FORT1 must be in Drive 1 to do so.*

10. WHAT

This command tells you the name of the workfile. Unless you use the GET command to change it, it will of course be FORT1:SYSTEM.WRK.TEXT.

11. BAD BLOCKS SCAN

If you suspect that your disk is faulty, you may check for possible damage with this command. It looks like this:

```
BAD BLOCK SCAN OF ? SID:   (check the diskette named SID)

SCAN FOR 280 BLOCKS (Y/N) ? Y   (check the entire disk)

0 BAD BLOCKS   (no damage was found)
```

If there *is* damage, you will see a message similar to this:

```
BLOCK 15 IS BAD

BLOCK 21 IS BAD
```

Later we will discuss another option which will attempt to "fix" bad blocks.

NOTE: Do not do a bad block scan of FORT2. FORT2 contains copy-protected files, and these will be interpreted as "bad blocks."

12. EXTENDED DIRECTORY LIST

This command gives additional directory information on files, including the location of free (unused) areas:

```
DIR LISTING OF ? FRED:

FRED:
MASTER.TEXT     23    3-APR-84   6   TEXT
MASTER.CODE     46    3-APR-84  29   CODE
<UNUSED>        50              75
TRANSFER.CODE   32    8-APR-84 125   CODE
<UNUSED>       123             157
3/3 FILES, 173 UNUSED, 123 IN LARGEST
```

This extended directory tells you that file TRANSFER.CODE is a CODE file, that it is 32 blocks long, and that it is stored beginning at block 125 (i.e., it is stored from block 125 to block 156).

13. KRUNCH

This rearranges disk storage so that all unused blocks are *together.* This allows one to save larger programs than could be saved on a non-Krunched disk. Unlike Applesoft BASIC's DOS, Apple FORTRAN's operating system always saves files in contiguous blocks. The above directory list is a Krunch candidate. Let's do it:

```
CRUNCH ? FRED:

FROM END OF DISK, BLOCK 280 (Y/N) ? Y

MOVING FORWARD TRANSFER.CODE

FRED --> CRUNCHED
```

Now when we list FRED's directory, we see at the bottom:

```
3/3 FILES, 173 UNUSED, 173 IN LARGEST
```

The used blocks have all been crunched together, as have the unused blocks!

14. MAKE

This command is used to create a directory entry for a non-existent file. It is seldom if ever used, and too detailed for inclusion here.

15. PREFIX

Recall that when a disk name is not specified, the operating system assumes that you refer to FORT1 (the boot diskette, containing Command Level). If you would like to change that assumption, do this:

```
PREFIX TITLES BY ? BERTHA:
```

Now you can refer to BERTHA:MAKE.TEXT simply as MAKE.TEXT.

16. VOLUMES

Informs you of the diskette names that are in the drives, and the output

and input devices you have available (you simply press **"V"**, and the operating system does the rest):

```
VOLS ON-LINE:
   1 CONSOLE:
   2 SYSTERM:
   4 FORT1:
   5 CHARLES:
   6 PRINTER:
ROOT VOL IS - FORT1:
PREFIX IS - BERTHA:
```

"SYSTERM" refers to the keyboard (SYStem TERMinal).
Volume 4 refers to Drive 1 (FORT1 is in Drive 1).
Volume 5 refers to Drive 2 (which contains disk CHARLES).
"ROOT VOL" refers to the disk used to boot FORTRAN.
"PREFIX" refers to the assumed disk name.

NOTE: THE VOLUME NUMBERS PROVIDE ANOTHER (SHORTER) WAY OF SPECIFYING OUTPUT DEVICES. Here are some sample uses (note the special symbol "#" which must precede the volume number):

```
TRANSFER ? MASS:MEDIA.TEXT
TO WHERE ? #1:          (transfer to the console)

TRANSFER ? BLANK:STARE.TEXT
TO WHERE ? #6:          (transfer to the printer)

REMOVE ? #5:FARMER.TEXT
              (remove from CHARLES:, which is in Drive 2)

DIR LISTING OF ? *      ("*" is shorthand for "root volume")

BAD BLOCKS SCAN OF ? :  (":" is used for "prefix volume")
```

17. XAMINE

This will attempt to patch up detected bad blocks. Electronic "damage" (a signal incompatible with Apple FORTRAN's operating system) may be "repaired" by Xamine since it re-formats *only the blocks you specify* (as opposed to the FORMATTER utility, which re-formats the *entire* disk). Physical damage (fingerprints, a scratch in the disk, etc.) *cannot* be repaired.

NOTE: The process of examining destroys the data contained in the specified blocks!

Recall that earlier we found that Blocks 15 and 21 of disk SID were bad. Let's patch them up:

```
EXAMINE BLOCKS ON ? SID:

BLOCK-RANGE ? 15-21

BLOCK 15 MAY BE OK
BLOCK 16 MAY BE OK
BLOCK 17 MAY BE OK
BLOCK 18 MAY BE OK
BLOCK 19 MAY BE OK
BLOCK 20 MAY BE OK
BLOCK 21 IS BAD
```

Now we know that Block 21 is beyond repair, but that 15 is probably "fixed." (Aside: Note that *all* blocks in the specified range are examined; it is legal to respond to the "BLOCK-RANGE?" prompt with a *single* block.) Now the option continues:

```
MARK BAD BLOCKS (FILES WILL BE REMOVED!) ?
```

Beware: If you mark the bad blocks, they will never be used. That's fine *except if the bad blocks are currently storing part of a file,* in which case *you will lose the entire file!* This command can therefore be both a blessing and a curse.

NOTE: As was stated before, do not scan or examine FORT2. If you "MARK" FORT2's "BAD-BLOCKS" (which aren't really bad), you will have lost the copy-protected FORTRAN Compiler. By the way, if you heeded the introductory section's advice and write-protected FORT2, such a disaster could not occur.

18. ZERO

This will wipe out your disk's directory. Luckily the process is loaded with "second chance" questions just in case you decide to back out:

```
ZERO DIR OF ? LUDWIG:

DESTROY LUDWIG: ? Y
```

```
DUPLICATE DIR ? N

ARE THERE 280 BLKS ON THE DISK (Y/N) ? Y

NEW VOL NAME ? LUDWIG2:

LUDWIG2: CORRECT ? Y

LUDWIG: --> ZEROED
```

You've just killed your disk!

C. REVIEW QUESTIONS

1. List the keystrokes necessary to make file GAS:MILEAGE.TEXT your new workfile.

2. You call in the Editor and find that someone forgot to wipe out his or her workfile. List the commands necessary to do so (no fair looking back to Chapter 1).

3. You would like to transfer all CODE files from disk WINDY to disk DRASTIC (and would not like to change any of their names). How would you do so?

4. Of what use is the Prefix command?

5. The Filer has four useful shorthand symbols (?, =, $, and volume numbers). Explain the meaning of each.

6. A person desires a printout of file "CHRISTMAS.TEXT" from his or her disk. When the person answers the "TRANSFER?" prompt, the Filer tells him or her "FILE NOT FOUND". What may have happened?

7. It was noted in the description of the Save command that suffixes should *not* be specified. What would happen if a person answered the "SAVE AS?" prompt with "ARMY:FLAG.TEXT"?

8. Can you think of a quicker way to destroy a disk's directory than the Zero command?

D. EXERCISES

1. Use the Editor to create some English files. Save them on your disk.

2. Use the Filer to:

   a. list your disk's directory.
   b. get a printout of one of the Text files.
   c. transfer each of the Text files to the console (hint: use a shorthand character).
   d. change one of the filenames.
   e. change the name of your disk.
   f. try as many other Filer options as time will allow (use volume numbers whenever possible).

Chapter 3:
COMPILER

A. FUNCTION

The Compiler is the most important component in program development. Its function is to translate a program which *you* can read and understand (in this case, FORTRAN) into a program which the *machine* can read and understand (machine language).

When you write a program in BASIC on the Apple, the BASIC Compiler translates *as you run the program*. Two problems are inherent in this strategy: one, it slows down program execution; and two, the translation process must take place again every time you re-run the program.

The FORTRAN Compiler operates in a more efficient manner. *Translation occurs before the program is ever executed, and it takes place only once (if we assume there are neither errors nor changes which will be needed).* As a result, FORTRAN programs are executed more rapidly than BASIC programs.

B. OVERLAYS

The Compiler is the largest of the Command Level files, and as such is written with overlays. Since the Compiler resides on FORT2, *this disk must be present in Drive 2 throughout the entire compilation process.*

C. COMPILER PROMPTS

After typing "C" from Command Level, you will see the following Compiler prompts:

1. COMPILE WHAT TEXT ?

NOTE: THE FILE READ IN AT THIS POINT MUST BE A TEXT (UNTRANSLATED, OR NON-MACHINE LANGUAGE) FILE. TO DISTINGUISH BETWEEN TEXT AND MACHINE LANGUAGE FILES, IT

IS STRONGLY RECOMMENDED THAT YOU USE THE SUFFIXES ".TEXT" AND ".CODE".

If you have used a workfile (FORT1:SYSTEM.WRK.TEXT), it will be read in automatically!

Otherwise, you will have to type a filename. The suffix ".TEXT" will be assumed, so you need only reply "FORMAT" to have file FORMAT.TEXT called in. *IF THE FILE IS NOT ON ONE OF THE SYSTEM DISKS, YOU WILL HAVE TO REMOVE FORT1 FROM DRIVE 1 AND REPLACE IT WITH THE DISK YOU SPECIFY IN YOUR FILENAME.*

2. TO WHAT CODEFILE ?

The Compiler is now asking, "What shall I call the machine language version of this program?" If you used a workfile, you have no need to answer; the translated program will automatically be called FORT1:SYSTEM.WRK.CODE (in fact, this prompt and the previous one won't even appear).

If you didn't use a workfile, you must supply a filename. It is recommended that you give your compiled version the same name as the text version, except with the suffix ".CODE". In fact, the Compiler will automatically supply this suffix, so you need only reply "BLANK:CHECK" to create the file BLANK:CHECK.CODE.

If you are wise enough to give the translated version the same name as the text version, you may simply type "$" (which means "use the same name as was used to call it in") in response to the prompt.

3. LISTING FILE ?

Once this prompt is answered, compilation will begin. Once it has, *DO NOT ATTEMPT TO STOP THE COMPILER BY PRESSING THE "RESET" KEY. This causes damage to the FORT2 disk! Be patient and let the Compiler complete its task (NOTE: Damage cannot happen if FORT2 is write-protected).*

Here are the two valid replies to the prompt:

a. Pressing RETURN

Most of the time, you will not need a Listing File, so you simply press the RETURN key. This causes your program's name to appear on the screen, followed by one dot for every program line successfully compiled.

NOTE: COMPILATION IS NEVER COMPLETE (I.E., A MACHINE LANGUAGE FILE IS NEVER PRODUCED) UNTIL NO ERRORS OCCUR DURING TRANSLATION. "WHY

TRANSLATE A FLAWED PROGRAM?" REASONS THE COMPILER.

If the Compiler detects errors, you will have to:

1. return to the Editor
2. edit your errors
3. quit the Editor and update the workfile
4. compile the corrected version

This cycle will often occur several times for every program you write, so don't get discouraged!

GEENEN'S LAW: A FORTRAN PROGRAM WILL NEVER COMPILE WITHOUT ERRORS ON THE INITIAL ATTEMPT. I HAVE NEVER SEEN AN EXCEPTION TO THIS RULE.

When you choose not to produce a Listing File, the Compiler reports errors (as soon as they are discovered) with a beep, a short message, and a new prompt:

```
***** ERROR NUMBER: 101 IN LINE: 31
<SP>(CONTINUE), <ESC>(TERMINATE), E(DIT
```

To find out what the error is, you will have to refer to the list of possible errors (all 405 of them!) in the Apple FORTRAN Language Reference Manual. *For help in deciphering these error messages, you may refer to Appendix B of this book, where many of the most common error messages are explained (in English).*

You may now either return to Command Level (ESC), continue compiling to see if any other errors were made (SPACE), or go directly back to the Editor (E). If you do return to the Editor and are using a workfile, it will be read in automatically; *you will then be positioned exactly two lines below the line containing the error* and told to type the spacebar to bring up the Editor prompt line.

b. Any valid filename

The Compiler can produce an elaborately documented version of the compilation process called a Listing File. The Listing File contains all program lines, error numbers (errors will *not* interrupt the compilation process when you choose to create a Listing File), available memory information, and a list of all variables and their corresponding types (Integer, Real, or Character). The Listing File may be sent to any disk file desired, but is most

often sent to the console (LISTING FILE ? #1:) or to the printer (LISTING FILE? #6:). For larger programs which need extensive debugging, the Listing File option can be quite useful.

D. EXAMPLES

1. You have saved your program in the workfile FORT1: SYSTEM.WRK.TEXT, and wish to compile it without creating a Listing File:

```
COMPILE WHAT TEXT ? (does not appear, answered automatically)

TO WHAT CODEFILE ? (does not appear)

LISTING FILE ? <RETURN>
```

2. You wish to compile program FORT1:CONNIE.TEXT in order to create file FORT1:CONNIE.CODE. You would like the Listing File for the compilation to appear on the console (note the use of the default disk name "FORT1", the default suffix ".TEXT", the "same name" shorthand symbol "$", and the volume number for "CONSOLE:"):

```
COMPILE WHAT TEXT ? CONNIE

TO WHAT CODEFILE ? $

LISTING FILE ? #1:
```

3. The file being compiled is BASE:TEN.TEXT, while the compiled version will be called BASE:TWO.CODE. A hard copy of the Listing File will be generated by the printer:

```
COMPILE WHAT TEXT ? BASE:TEN

TO WHAT CODEFILE ? BASE:TWO

LISTING FILE ? #6:
```

In this case, disk BASE must be placed in Drive 1 before the compilation.

When at last your program compiles without errors, you will be returned to Command Level (if you earlier removed FORT1, you will now have to place it back in Drive 1 for Command Level to reappear).

YOU ARE NOW READY FOR THE LINKING PROCESS, THE FINAL STEP BEFORE EXECUTION.

E. REVIEW QUESTIONS

1. Do you think that, when first compiling a large program, you would be wise to call in the Editor each time a mistake is reported?

2. Let's say a person opts to let the Compiler find all the program errors before calling in the Editor. Errors are reported in lines 34, 56, and 86. How may these lines be located?

F. EXERCISES

1. Type the following program which converts a fraction to its decimal equivalent. Make several errors while doing so, then try to compile it. (NOTE: Even though you were instructed earlier to type "R" for "R(UN", at this point I would suggest typing "C" to emphasize the compiling process. When you do so, your program will be neither linked nor executed.) Become as familiar as you can with the process of checking reported error numbers, calling the workfile back into the Editor to make the necessary corrections, updating the workfile, and compiling once again. This is a crucial skill, and one that can never be "over-practiced." You should also become familiar with the process of creating a Listing File during this exercise.

```
      PROGRAM CONVRT
      INTEGER DENOM
      DATA NUMRTR, DENOM / 1781, 1Ø6 /

      DECIML = REAL (NUMRTR) / DENOM
      WRITE (*, 1Ø) 'The rounded decimal equivalent of ',
     *              NUMRTR, ' / ', DENOM, ' = ', DECIML
1Ø    FORMAT (A, I4, A, I3, A, F6.3)

      END
```

2. Try compiling a text file on your disk *without* creating a workfile. Even though you will seldom do so in practice, it will serve two purposes: one, it will give you a better understanding of the Compiler and its prompts; and two, it will help you appreciate more fully the benefits of using a workfile. Be prepared to do plenty of disk shuffling during this exercise.

3. The Editor and Filer will operate on files written in *any* language (i.e., they are *language-independent*). It is the Compiler which insists on accepting only files written in FORTRAN. If you have time, you may convince yourself of this fact by trying to compile a file that you have written in BASIC or English.

Chapter 4:
LINKER

A. FUNCTION

The Linker is the last step in program development before actual execution. It can literally link independently compiled programs; in other words, two or more partial programs can be pieced together to form a complete whole. We will learn how to do this in a later chapter. So why describe the Linker now? Because *every* program must be linked with the System Library! The System Library contains some machine language units which must be "attached" to a program before it can run.

B. NO OVERLAYS

The Linker, like the Filer, is a comparatively small file, written with *no overlays* (read in its *entirety* into the computer's memory). It is contained on disk FORT2.

C. LINKER PROMPTS

1. HOST FILE ?

If you simply press RETURN in response to the above prompt, the file FORT1:SYSTEM.WRK.CODE will be read in as the program to be linked. Otherwise, you will need to supply a filename. *This file must be a machine language file.* If you respond with "CHANGE", the Linker will read in CHANGE.CODE (there is no need to type the ".CODE" suffix).

2. LIB FILE ?

Recall that you must link the System Library with every program before its execution. You should therefore type "*" *(no quotes!)* in response to this prompt. This is short for the file FORT1:SYSTEM.LIBRARY, so FORT1 must be present for linking to take place.

3. LIB FILE ?

This may seem redundant, but the repetition of this prompt allows you to link several programs. For now, however, just press RETURN (which means "no further Library files").

4. MAP FILE ?

A Map File is an advanced programming tool, and in general is not very helpful even if you have the ability to understand it. Since we will not be using Map Files, just press the RETURN key.

The Linker will now inform you which Code programs it will attempt to link. You'll see seomething like this:

```
READING MAINSEGX     (this is part of FORT1:SYSTEM.LIBRARY)
READING CHANGE       (this is the name of your program)
READING RTUNIT       (this is part of FORT1:SYSTEM.LIBRARY)
```

Only two subunits (MAINSEGX and RTUNIT) of FORT1:SYSTEM. LIBRARY are linked into the program. The other units in the library are used for random number generation, graphics, and sound generation (more on these other units later in the book).

Finally, the Linker asks:

5. OUTPUT FILE ?

Responding by pressing RETURN saves the linked version under the name SYSTEM.WRK.CODE (on FORT1, as are all other versions of the workfile). Those not using a workfile will need to respond with a filename. It is suggested that you give the linked version of your program the same name as the non-linked version. This wipes out the non-linked version, thereby saving precious disk space.

Linking will now take place, after which you will be returned to Command Level. You're finally set to execute your program!

"Are there any possible errors inherent in the linking process?" you ask. Yes, but fortunately only one:

```
CODE WRITE ERROR
```

This means that there isn't enough room for the output file on the specified disk. You may have to go back to the Filer and delete some files (or possibly "Krunch" the disk) if room is not available. Then you will have to repeat the linking process from the start.

D. EXAMPLES

1. You would like to link your compiled workfile FORT1: SYSTEM.WRK.CODE:

```
HOST FILE ? <RETURN>
LIB FILE ? *
LIB FILE ? <RETURN>
MAP FILE ? <RETURN>
OUTPUT FILE ? <RETURN>
```

2. You would like to link the previously compiled file CLIFF:SWALLOW.CODE with the System Library. The linked version will be given the name CLIFF:HANGER.CODE. You're dying to see what the Linker's Map File looks like, so you'll send it to the console:

```
HOST FILE ? CLIFF:SWALLOW
LIB FILE ? *
LIB FILE ? <RETURN>
MAP FILE ? #1:
OUTPUT FILE ? CLIFF:HANGER
```

At the beginning of the operation, FORT1 and FORT2 would be on-line. After you pressed "L" to commence linking, FORT2 would need to be removed to make way for disk CLIFF (recall that this is possible since the Linker, which resides on FORT2, has no overlays). FORT1 contains the System Library and, as a result, needed to remain in Drive 1.

E. REVIEW QUESTION

Why would one want to join two separately compiled programs with the Linker?

F. EXERCISE

In the previous chapter, you were asked to compile a program contained on your disk without the aid of a workfile. If you were able to do so, try to link the now-compiled program in the same manner. This will require you to answer all the Linker prompts given in this chapter. As before, you will gain a greater understanding of the Linker by doing so, even though you will always use workfiles in practice.

Chapter 5:
EXECUTION AND SUMMARY

A. EXECUTING YOUR PROGRAM

1. Executing a Compiled Program

Now that you've compiled and linked your program, you're finally ready to execute it!

NOTE: To run your program, type "X" (for execute), not "R" (for run). "RUN" means something quite different, and will be explained later in this lesson.

2. Prompt

```
EXECUTE WHAT FILE ?
```

You *must* respond with a filename, whether or not you have a workfile. Recall that if no disk name is supplied, the operating system will assume you mean the boot diskette FORT1. The disk containing the program should of course be in one of the drives. The suffix ".CODE" will be assumed.

Now you're home free, right? *Wrong!* Even though you've made it past the Compiler with no errors, you may still get Run-Time errors. In fact, the *Language Reference Manual* contains a list of 100 possible errors. Check out Appendix B if you have any trouble deciphering these messages.

What does one do if one encounters a Run-Time error?

a. Punt.

b. Pray for divine intervention.

c. Return to the Editor and patch up errors.

d. Compile, Link, and Xecute again.

Eventually, however, you will have a working program. At this stage, you should save *two* versions of your program:

1. The text version. This is in the event you ever decide to alter the program. *You cannot edit a code file.*

2. The code version. This is the file you will execute. It is the already

compiled and linked machine language program that you worked so hard to produce.

B. THE "RUN" SHORTCUT

It is extremely likely that you will have to go through the Compile-Link-Xecute sequence several times in the course of creating your programs. Fortunately, the Apple FORTRAN operating system has a very useful shortcut for our benefit.

Once you have your program created by the Editor and saved in the workfile SYSTEM.WRK.TEXT, just type "R" (for "RUN") from Command Level. This option will automatically Compile, Link, and Xecute the workfile (unless of course the Compiler finds errors, whereupon you must return to the Editor).

Following are the operating system prompts that need to be answered when you choose the Run option, and appropriate responses for two situations.

SITUATION 1: You have just saved the text version of your program in the workfile (FORT1:SYSTEM.WRK.TEXT).

SITUATION 2: Your program was saved as BAD:JOKE.TEXT.

PROMPT	SITUATION 1	SITUATION 2
COMPILING. . .		
COMPILE WHAT TEXT ?	automatic, SYSTEM.WRK.TEXT	BAD:JOKE
TO WHAT CODEFILE ?	automatic, SYSTEM.WRK.CODE	$
LISTING FILE ?	<RET>, or #1:, or #6:	<RET>, or #1:, or #6:
LINKING. . .		
HOST FILE ?	automatic, SYSTEM.WRK.CODE	BAD:JOKE
LIB FILE ?	automatic, *	*
LIB FILE ?	automatic, <RET>	<RET>
MAP FILE ?	automatic, <RET>	<RET>
OUTPUT FILE ?	automatic, SYSTEM.WRK.CODE	BAD:JOKE
RUNNING. . .		
EXECUTE WHAT FILE ?	automatic, SYSTEM.WRK.CODE	BAD:JOKE

Notice how much typing you are spared by using a workfile and the Run option! The *only* prompt you have to answer is "LISTING FILE?". (In fact, it is the only prompt you will even *see!*)

To further convince you of the benefits of using a workfile, a table showing the various disks which must be present during the RUN process follows:

OPERATION	SITUATION 1	SITUATION 2
COMPILING	FORT1, FORT2	BAD, FORT2
return to Command Level	FORT1, FORT2	FORT1, FORT2
Linker read in	FORT1, FORT2	FORT1, FORT2
LINKING	FORT1, FORT2	FORT1, BAD
EXECUTION	FORT1, FORT2	FORT1, BAD

Notice that there is no disk shuffling associated with workfile use.

C. SUMMARY OF APPLE FORTRAN DISKS

Recall that those Command Level options written with overlays must remain in a drive for the duration of their implementation.

FORT1	FORT2
COMMAND LEVEL	COMPILER (overlay)
SYSTEM LIBRARY	EDITOR (overlay)
WORKFILES	FILER
	LINKER
	FORMATTER

It would be very wise to place labels similar to the above on the two system disks.

D. SUMMARY OF PROGRAM DEVELOPMENT

This summary assumes you have just booted FORTRAN with FORT1 in Drive 1 and FORT2 in Drive 2. You have a program which you wish to create and execute.

1. From Command Level, type **"E"** to enter the Editor. Since the program is new, no workfile should be read in. (If a workfile is read in, it must be someone else's. **Q(UIT** the Editor; **E(XIT WITHOUT UPDATING;** enter the **F(ILER;** choose the **N(EW** option, which will wipe out the old workfile when you tell it **Y(ES—WIPE OUT OLD WORKFILE; Q(UIT** the Filer; and call the **E(DITOR** once again.) Press the **RETURN** key when

asked for a file name. (The file *has* no name yet, it hasn't even been typed!) You will now see the Editor prompt line.

2. Type your program (remember to start with **"I"** for "INSERT"). When finished *(don't forget CTL-C to make it permanent),* **Q(UIT** the Editor and choose to **U(PDATE THE WORKFILE.** Your file has now been saved as FORT1:SYSTEM.WRK.TEXT.

3. From Command Level, choose the **R(UN** option. Compiling, Linking, and Xecution will now occur. The only prompt you will see is the Compiler's "LISTING FILE?", which is usually answered by pressing **RETURN.** Everything else is automatic!

4. If no errors occur (or you have no desire to revise anything), you've finished! Look ahead to step 6 for instructions on saving your program on your own disk.

5. If you're mortal, you'll have to go back to the old drawing board. Call in the **E(DITOR.** It expects you back, so it will automatically read in your workfile. Make your corrections and/or revisions (hopefully without taking it out on the computer by pulling out all of its circuits). When done, **Q(UIT** the Editor, **U(PDATE THE WORKFILE,** and try **R(UN**ning again. Repeat this procedure until you've gotten it right.

6. You've made it! Congratulations! You now have the files SYSTEM.WRK.TEXT and SYSTEM.WRK.CODE on FORT1, and would like to transfer them to your own disk. Call in the **F(ILER,** and choose its **S(AVE** option.

Since the Filer now resides entirely in memory, remove FORT2 from Drive 2 and replace it with your disk (let's say it's called "SPECIAL"). Here's what you should do if you want to call your file "FRIEND" (your reply is **emphasized,** as usual):

```
SAVE AS ? SPECIAL:FRIEND

SYSTEM.WRK.TEXT --> SPECIAL:FRIEND.TEXT

SYSTEM.WRK.CODE --> SPECIAL:FRIEND.CODE
```

You can now **L(IST DIRECTORY** of your disk to ensure that the saving process worked.

Last but not least, use the Filer's **N(EW** option to erase both workfile versions so the FORT1 disk is ready for the next person to use.

E. EXECUTING A PROGRAM ON YOUR DISK

This is easily done. Remove FORT2 and replace it with your disk. Now type **"X"** (not "R") while at Command Level, and respond to the prompt with **"SPECIAL:FRIEND"** (the .CODE suffix is automatically supplied), or some other filename.

F. EDITING A PROGRAM ON YOUR DISK

Since the file is not a workfile, you must supply the filename when you call in the Editor. Here's how it might look:

```
NO WORKFILE IS PRESENT. FILE? ( <RET> FOR NO FILE, <ESC-RET> TO EXIT)
: FRENCH:FOOD
```

The Editor will automatically supply the ".TEXT" suffix. Don't forget to replace FORT1 with FRENCH before answering this prompt! Once the file has been read in, replace FRENCH with FORT1.

Now proceed to edit as usual. When you've finished, **Q(UIT** the Editor and choose to **U(PDATE THE WORKFILE.** This of course will save your file as FORT1:SYSTEM.WRK.TEXT. You may now proceed as usual with a workfile.

CONGRATULATIONS! You have now learned all there is to know about the Apple FORTRAN operating system. On to the FORTRAN language itself!

G. REVIEW QUESTIONS

1. Why is it necessary to save two versions of a completed program?

2. List all stages of program development that are made easier with the use of a workfile and the R(UN command.

3. Which Command Level options are written with overlays?

4. What would happen if a person typed "R(UN" instead of "X(ECUTE" when trying to execute a program already saved on her/his disk?

H. EXERCISES

1. Type the gas mileage program you were given in the Introduction.

Using a workfile, generate an executable Code file. Save both versions of the finished program on your disk.

2. Now we're going to simulate your coming back to the computer a week later to make some changes. Place the system disks in their proper drives and re-boot by pressing CTL-RESET. Bring your program from "last week" into the Editor. Make the changes described in the Introduction (i.e., change the two lines to "GALLNS = 17.5" and "DO 40 MILES = 300, 600, 25"). Finally, save the program as your workfile and get it running. When finished, save the Text and Code of the new version on your disk.

3. Lastly, we will simulate your executing the program a week later. Again re-boot with CTL-RESET. Execute the gas-mileage program you saved "last week." Repeat exercises 2 and 3 as often as time allows; familiarity with these operations will prove invaluable in the future.

I. OPERATING SYSTEM COMMAND SUMMARY

SPECIAL KEYSTROKES:

CTL-@ user break key
CTL-A flips between left- and right-screen pages (40-column machines)
CTL-Z automatically follows cursor left or right (40-column machines)
CTL-F flushes program output until the next CTL-F
CTL-S stalls program output until the next CTL-S

COMMAND LEVEL—FORT1

```
COMMAND : E(dit, R(un, F(ile, C(ompile, L(ink, X(ecute, A(ssemble, D(ebug, ?[1.1]
```

EDITOR—FORT2 (overlays); assumed suffix is ".TEXT"

```
No workfile is present. File? ( <ret> for no file, <esc-ret> to exit)

>Edit : A(djst C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(change Z(ap

Adjust : L(just R(just C(enter <left,right,up,down-arrows> [<ETX> to leave]

Delete : < > <moving commands> [<ETX> to delete, <ESC> to abort]

>Find [1] L(it <target> =>

Insert : Text [<BS> a char, <DEL> a line] [<ETX> accepts, <ESC> escapes]

Jump : B(eginning E(nd M(arker <esc>
```

```
>Replace [1] L(it V(fy <targ> <sub> =>

Quit :
     U(pdate the workfile and leave
     E(xit without updating
     R(eturn to the editor without updating
     W(rite to a file name and return
     S(ave with same name and return

Exchange : Text [<BS> a char] [<ESC> escapes; <ETX> accepts]
```

FILER—FORT2; no assumed suffix; shorthand characters "=", "?", and "$"

```
G, S, N, L, R, C, T, D, Q          W, B, E, K, M, P, V, X, Z

     Save      Save as?
```

(note : do *NOT* supply a suffix for this option only)

```
     New : Throw away current workfile?

     List : Dir listing of?

     Remove : Remove?
              Update directory?

     Change : Change?
              Change to what?

     Transfer : Transfer?
                To where?

     Date : Today is 2-JUL-84
            New date?

     Bad blocks scan : Bad blocks scan of?
                       Scan for 280 blocks (Y/N)?

     Extended directory list : Dir listing of?

     Krunch : Crunch?
              From end of disk, block 280 (Y/N)?

     Prefix : Prefix titles by?

     Volumes      Volumes on line :
```

(followed by list of volume numbers and devices)

```
Xamine : Examine blocks on?
         Block-range?
         Mark bad blocks (files will be removed!)?

Zero : Zero dir of?
       Destroy (disk name)?
       Duplicate dir?
       Are there 280 blks on the disk?
       New vol name?
```

COMPILER—FORT2 (overlays); shorthand character "$"

```
Compile what text?
```
(assumed suffix is ".TEXT")

```
To what codefile?
```
(assumed suffix is ".CODE")

```
Listing file?
```

LINKER—FORT2; assumed suffix is ".CODE"

```
Host file?

Lib file?

Lib file?

Map file?

Output file?
```

J. OPERATING SYSTEM COMMAND TREE

See the following pages.

Following is the Apple FORTRAN Operating System Tree. Along the left edge are the Command Level options; these represent the outermost level of the tree. From each level, you will see two arrows, one to move you into that option, and one to move you out of it. The inward arrow describes the keystrokes necessary to recover from *accidentally* choosing that option. Normally, one exits an option by answering its prompts, or by selecting an option from its submenu. For example, if I *accidentally* choose option R(place from the E(dit prompt line, I may recover by pressing the <esc> key. However, if I *want* to R(place, I merely enter a target and a substitute string, after which I will *automatically* be returned to the Edit prompt line.

EDIT

E →
← if workfile is read in : Q E
if no workfile read in : <esc> <ret>

A(djst	C(py	D(lete	F(ind	I(nsrt	J(mp	R(place	Q(uit	X(change
Ctl-C ⇅ A	<esc> ⇅ C	<esc> or ⇅ D Ctl-C	<esc> ⇅ F	<esc> or ⇅ I Ctl-C	<esc> ⇅ J	<esc> ⇅ R	R ⇅ Q	<esc> or ⇅ X Ctl-C
L)just	not used yet	cursor moving keys	/target/	text	B(eginning	/targ/	U)pdate	text
R)just					E(nd	/subst/	E)xit	
C)enter					M(arker		R)eturn	
							W)rite	
							S)ave	

RUN

R →
← <esc> <ret> <esc>

equivalent to C(ompile + L(ink + X(ecute; if workfile is present, all prompts answered automatically, except Compiler's "Listing file ?"

Figure 5.1 Operating System Command Tree (continued)

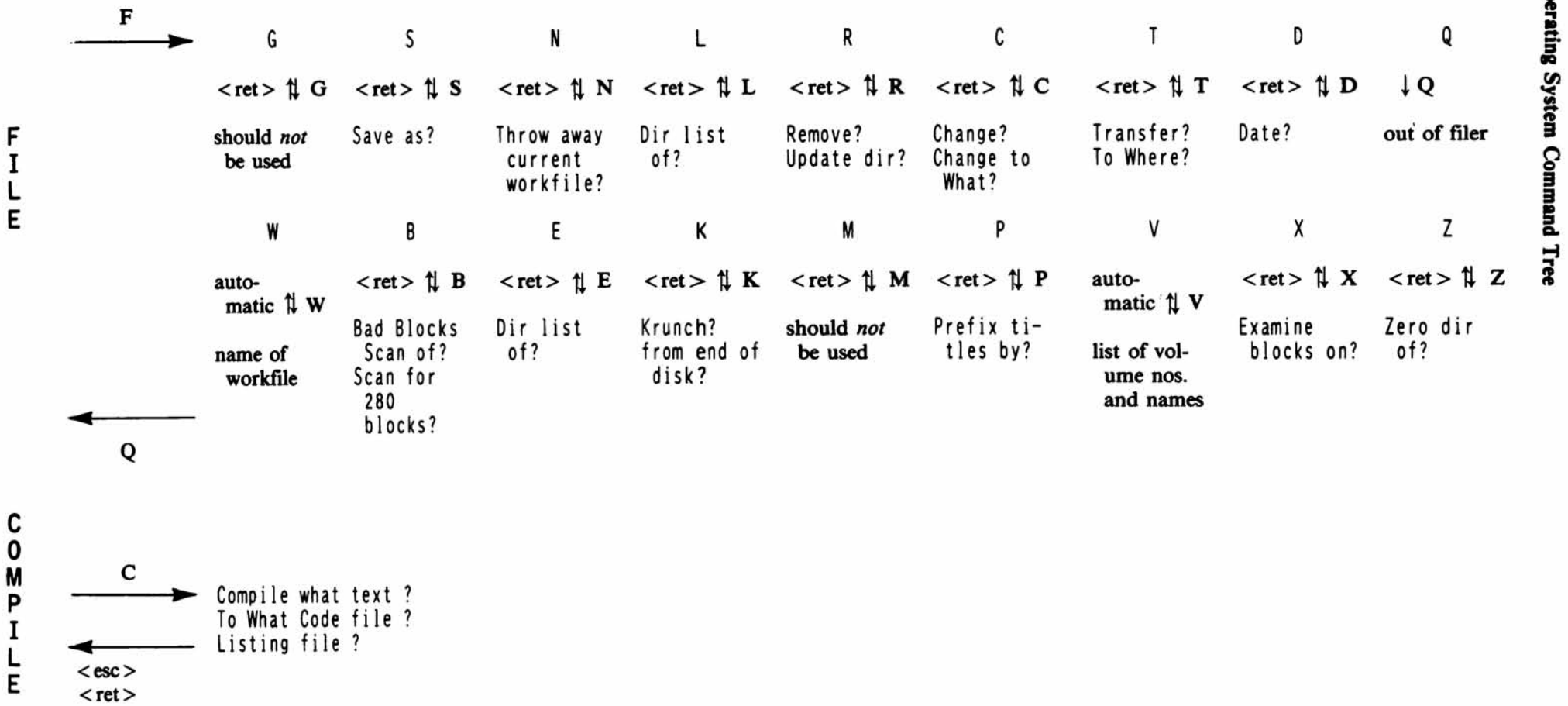


Figure 5.1 Operating System Command Tree (continued)

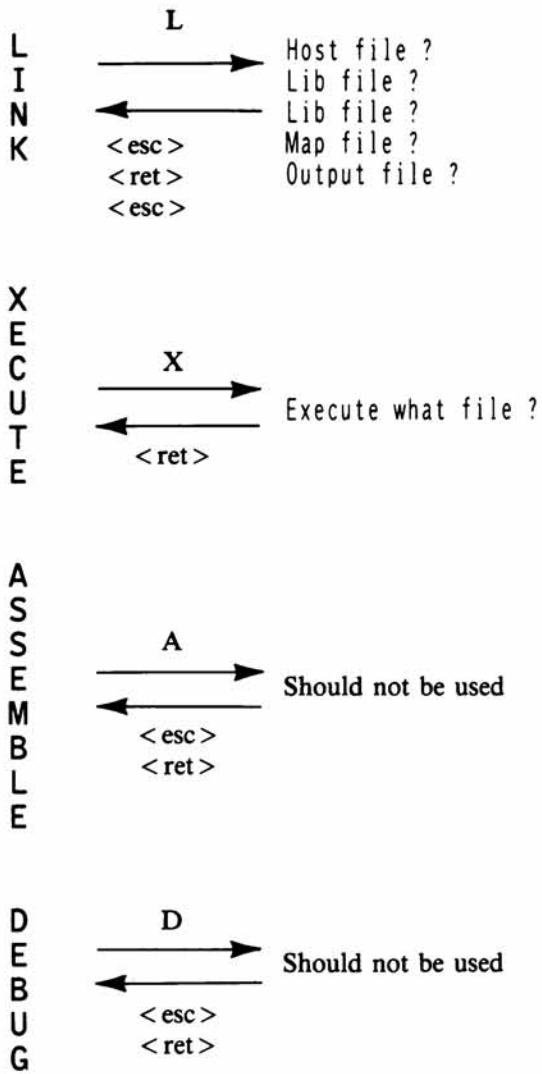


Figure 5.1 Operating System Command Tree

Part II
THE FORTRAN LANGUAGE

Chapter 6:
INTRODUCTION TO FORTRAN

Now that we've learned how to write, develop, alter, and save programs via the operating system, it's time for us to begin learning the actual FORTRAN language. Oh yes, in case you're wondering, FORTRAN is short for FORmula TRANslator.

A. STATEMENT STRUCTURE

Originally, FORTRAN lines were read from punched cards. This resulted in some rather strict rules about what can be placed where in a program line. Be forewarned that even modern-day FORTRAN compilers enforce the old rules! Here they are:

```
AAAAABCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCDDDDDDD
```

1. Columns 1–5, marked with A's above, are reserved for line numbers. *Line numbers are not required for most FORTRAN lines; if a line number is not used, you must leave these columns blank.* Statements are normally executed in the order in which they appear.

2. Column 6, marked with a B, is reserved for a continuation character (an asterisk-*). This is used to signal a continuation of the previous line. (We are about to see that FORTRAN allows a maximum of seventy-two characters per line.)

Column 6 MUST be left blank in all other lines. In addition, a continuation line may not have its own line number (Columns 1–5 must be blank). A single program line may contain up to nine continuation lines.

3. The actual instructions must be placed in Columns 7–72 (the C's above). Blanks are *not* significant in FORTRAN unless they are part of a character string. Therefore, a statement may begin after column 7. This allows us to use TAB (or CTL-I) to begin a line. (TAB actually moves you to column 9, but it sure beats counting out six spaces before typing an instruction!) It also allows indentation of loops, decisions, etc. Yet a line may never extend beyond column 72. If a line is too long for the 72-character limit, one must

use a continuation line (see 2 above). Apple FORTRAN's Editor beeps as a reminder when column 72 is reached.

4. Columns 73–80, marked with D's above, were originally used to write comments on the punched cards (usually line numbers to keep the cards in proper order). They are not used in Apple FORTRAN. Once you hear the beep, characters will continue to be accepted by the Editor, but *ignored* by the Compiler. Finally, if you attempt to enter characters past the 80th column, an exclamation point will appear, and no further text will be accepted.

End all lines by pressing the RETURN key.

NOTE: These rules must be followed in every single line of a program or else that program will never be successfully compiled.

B. VARIABLES AND DATA TYPES

Variables in FORTRAN are similar in nature to variables in other programming languages: they are named memory locations which may store exactly one value at a time; if a variable is given a new value, its old value is lost. In FORTRAN, variables must begin with a letter (A–Z), followed optionally by letters and/or digits (0–9).

THE MAXIMUM NUMBER OF CHARACTERS ALLOWED IN A FORTRAN VARIABLE IS 6. This restriction causes one of FORTRAN's major readability problems, as it may result in some rather strange or even humorous abbreviations. ("TAXRATE" may turn into "TAXRAT"—ouch!) Yet a creative programmer will usually be able to come up with reasonable variable names. (How about "TXRATE" as an alternative?) (Aside: It always puzzled me why a language which only allows *six*-letter variables would have a *seven*-letter name! Perhaps we should rename the language FORTRN or FRTRAN.)

Here are some *valid* variables:

```
YEARS

MAX

AVE

A123

Q
```

NOTE: THERE ARE NO RESERVED WORDS IN APPLE FORTRAN. (Keywords are interpreted by their position or context in a program line.) SO THERE ARE NO RESTRICTIONS ON VARIABLE NAMES. IN ADDITION, ALL SIX CHARACTERS OF A VARIABLE NAME ARE SIGNIFICANT. ("DELAY1" and "DELAY2" are DIFFERENT variables.) THIS IS QUITE UNLIKE THE SITUATION IN APPLESOFT BASIC (which has several reserved words and only two significant characters in its variables)!

Here are some *invalid* variables:

```
1A        (begins with a number)

AVERAGE   (7 characters)

ME&YOU    (the ampersand is neither a letter nor a digit)
```

The first thing a programmer must do when writing a FORTRAN program is name it. This is done in the *first line* of the program by using the keyword PROGRAM followed by any valid variable. (Aside: Apple FORTRAN will allow you to omit the PROGRAM designation, whereupon it will call your program NONAME.) Don't forget about program lines beginning no earlier than column 7! Examples:

```
PROGRAM GASUSE

PROGRAM CHANGE

PROGRAM VEGAS
```

NOTES:

1. *Once your program has been named, don't attempt to use a variable with the same name as the program. For instance, in the second example above, the programmer is not free to use "CHANGE" as a variable, since "CHANGE" has already been used to name the program.*

2. *The last thing a programmer must do when finished with a program is type the reserved word "END" followed by exactly ONE press of the RETURN key. THE WORD "END" MAY NEVER APPEAR ANYWHERE EXCEPT AS THE LAST LINE OF A PROGRAM.*

1. Integer Data

a. Any number containing neither a decimal nor an exponent is considered

an integer in FORTRAN. Integers are also limited to values in the range −32,768 to +32,767.

b. Integers require two bytes of storage.

c. Any variable *beginning* with the letters I–N can store Integers *only!* If you attempt to store a Real number with an Integer variable, the INT (decimal-truncation) function will be called in automatically.

2. Real Data

a. Any number containing a decimal or an exponent or having an absolute value larger than 32,767 is considered a Real (or Floating-Point) number in FORTRAN.

b. A Real number requires four bytes of storage (twice as much as an Integer).

c. A variable *beginning* with the letters A–H or O–Z can store Real numbers *or* Integers. *(NOTE: IF YOU DO STORE AN INTEGER WITH A REAL VARIABLE, IT WILL STILL USE FOUR BYTES OF STORAGE. This is because the Integer will be converted to its Floating-Point equivalent by the REAL function; i.e., 125 will be automatically converted to 125.Ø.)*

d. Apple FORTRAN can store real numbers in the range 1E+38 to 1E-38. They are stored with a six significant-digit precision.

3. Character Data

a. The maximum length a character string may attain in FORTRAN is 255 characters.

b. Character variables require one byte of storage for each character.

c. If Integer and Real variables are determined by default (using the first letter of their names), and the default covers all 26 possible letters, is there anything left for Character variables? Yes! *All Character variables must be declared in the beginning of the program (immediately after the program has been named). The declaration overrides the Integer or Real designation the variable would otherwise have.*

Here is the format for character variable declaration:

```
CHARACTER variable*maximum length
```

NOTE: Beware that the Compiler expects a "PROGRAM" statement as the first line of a program, followed by any declared character variables, and then the instruction lines; the last line of the program must be "END" followed by a SINGLE *"RETURN" press. If you mix up this order, you will never successfully compile the program!*

Here's a look at an acceptable program's beginning:

```
PROGRAM WAGES

CHARACTER WORKER*30
```

WORKER is now a Character variable which may store up to 30 characters. It requires 30 bytes of storage.

NOTE: If a Character variable is asked to store a string with FEWER characters than its declared maximum length, the operating system will add trailing blanks to fill out the string. If you assign a character variable a string with MORE characters than its declared maximum length, only the LEFT-MOST characters (those within the specified limit) will be stored.

Here's how to declare more than one Character variable at a time:

```
PROGRAM WEATHR

CHARACTER STATN*25, MONTH*3
```

If no Character variables will be used, you may of course skip the CHARACTER statement.

C. ASSIGNMENT STATEMENTS

1. Operators

+ addition
− subtraction
* multiplication
/ division
** exponentiation (NOTE: a caret, ∧, will not work)

The ordering of operations is standard algebraic; first parentheses, then exponents, followed by multiplication and division (which have equal standing), and finally addition and subtraction (which have equal standing); all operations of equal priority are performed left to right.

2. Form

```
variable = expression
```

NOTE: The word "LET" does not exist in FORTRAN.

Examples:

```
AREA = 3 * WIDTH

PERCNT = (COMPLT / ATMPTS) * 100

CHANGE = MONEY - PRICE

C = (A ** 2 + B ** 2) ** 0.5
```

NOTES:

1. *FORTRAN will not assume a value of zero for non-assigned variables! You must explicitly assign values to all variables.*

2. *FORTRAN will not allow multiple statements per line.*

3. *Apple FORTRAN has no "CLEAR" command.*

Armed with the above knowledge, how can one quickly assign several variables a value of Ø (or any other value) without using one program line per variable? With a modified assignment statement called "DATA". Here is its form:

```
DATA list of variables / list of values /
```

You may *not* use variables or expressions on the right side of DATA.

Some examples will help you understand the form:

```
DATA A, B, C, D / 12.3, 134.2, 10.0, -10.3 /
```

This assigns A the value 12.3, B the value 134.2, C the value 1Ø.Ø, and D the value −1Ø.3.

```
DATA I, J, K / 10, 15, 20 /
```

This is equivalent to the following program segment:

```
I = 10

J = 15

K = 20
```

It is also possible to assign character variables their contents with DATA:

```
DATA NAME, RANK, SERIAL / 'Joe Computer', 'Major', 'Q4567' /
```

Notice the single quotes.

There is no limit to the number of variables you may assign in this manner. Just be sure that each variable has a corresponding value.

NOTES:

1. *You must be sure that the type (Integer, Real, or Character) of the variable and its value match precisely.*

2. *"DATA" must be placed before any executable lines.*

Following is a more visual description of statement ordering:

PROGRAM	(always first)
CHARACTER	(optional, second if present)
DATA	(optional, third on the "ladder")

3. Type Conversions

Recall that FORTRAN distinguishes between Integer and Real variables. What would happen if a person needed an expression using both Integers and Reals? This table summarizes the Type conversions employed within assignment statement expressions in FORTRAN:

OPERANDS	RESULT
two Integers	INTEGER
two Reals	REAL
one Real, one Integer	REAL

NOTE: Type conversions often cause unexpected results if you are not aware of them. Study the following assignment statements carefully:

STATEMENT	VALUE STORED	COMMENTS
`I = 23.9`	23	"INT" called in
`A = 12`	12.Ø	"REAL" called in
`J = 13 / 3`	4	dividing two integers!
`B = 13 / 3`	4.Ø	why not 4.333333?
`C = 13.Ø / 3`	4.333333	see the difference?
`K = 12.Ø + 3.6`	15	"K" can only store integers
`D = 7 + 4`	11.Ø	"11" would be incorrect!
`L = (3 + 4) * 1.5`	1Ø	right side yielded 1Ø.5

The examples have attempted to illustrate these rules:

TO EVALUATE AN ASSIGNMENT STATEMENT CORRECTLY:

a. *Evaluate the expression (right side of the equation) first.* Always operate on only two values at a time (remember the order of evaluation), and be aware of the type conversions FORTRAN employs.

b. *The variable (left side of the equation) has final say over what is stored.* Integer variables can store only Integers, Real variables only Reals. If the variable doesn't match the type of the value returned from the right side, apply either INT (decimal truncation, *not rounding*) or REAL (conversion to Floating-Point equivalent). Assignment statements are therefore different from "DATA" statements, for "DATA" will *not* apply "INT" or "REAL" for you.

4. Assigning Character Variable Contents

a. The form was alluded to in the above description of the DATA statement:

```
character variable = 'string'
```

b. Examples

```
NAME = 'Joseph J. Computer'
ADDRSS = '2105 E. Forest St.'
```

Notice the single quotes. NAME and ADDRSS would have been declared as Character variables at the beginning of the program, of course.

A special problem arises when one tries to store a single quote (or apostrophe) as a character. This is done by listing *two immediately adjacent single quotes:*

```
BOOK = 'Let''s Go Fishing'
```

c. It is also possible to store the contents of one Character variable in another. Here is an example:

```
NAME1 = NAME2
```

The contents of NAME2 would now be stored in NAME1.

d. One cannot concatenate ("add") Character variables in FORTRAN.

D. REVIEW QUESTIONS

1. Discuss the format of a FORTRAN program line.

2. How does one declare variables for each of the three data types?

3. What are the first and last lines of any FORTRAN program?

4. Of what use is the DATA statement? Why do you think it must be placed after the CHARACTER declaration?

5. For each of the following assignment statements, list all the steps necessary to determine the value which will be stored:

a. `AREA = 6`

b. `INDEX = 28.9`

c. `VALUE = 4.0 / 2`

d. `ANSWER = (4 * 3.0) ** 2`

e. `LENGTH = 4.0 * 2.0 / 10.0`

f. `FINAL = 27 / 4`

6. What would happen if a programmer entered the following assignment:

```
BOOK = 'Let's Go Fishing'
```

E. EXERCISES

It was not possible for you to understand much of the content of the sample programs given in Chapters 1 through 5. This is only natural, since these chapters dealt with the operating system, and we have only now begun to study the FORTRAN language. However, I would like to make it clear that from now on, the sample programs are intended to reinforce previously covered material. You must review those program segments with which you are unfamiliar! (That being said, I would like to add one minor qualification—you have not yet studied the WRITE and FORMAT commands, so you cannot fully comprehend the following program. This is your *last* exemption.)

1. Enter and R(UN the following program:

```
PROGRAM PASSES
CHARACTER QRTRBK*20
```

```
      QRTRBK = 'Joe Fortran'
      ATMPTS = 255.Ø
      CMPLTS = 157.Ø

      WRITE (*, 1Ø) 'Quarterback : ', QRTRBK
1Ø    FORMAT (/, A, A)
      WRITE (*, 2Ø) 'Attempts : ', ATMPTS
2Ø    FORMAT (/, A, F5.1)
      WRITE (*, 2Ø) 'Completions : ', CMPLTS

      PERCNT = (CMPLTS / ATMPTS) * 1ØØ
      WRITE (*, 2Ø) 'Percentage : ', PERCNT

      END
```

2. Make the program shorter by using DATA to assign values for QRTRBK, ATMPTS, and CMPLTS. R(UN this new version.

3. Try to make as many of the following mistakes as you possibly can. Make each of the mistakes *individually,* then try to compile. When the Compiler stops you, correct the error and enter the next mistake given on the list:

a. name your program "QRTRBK".

b. omit the CHARACTER declaration.

c. omit the single quotes around "Joe Fortran".

d. change the single quotes to double quotes.

e. change "255.Ø" to "255".

f. use "ATTEMPTS" instead of "ATMPTS".

g. use "COMPS" instead of "CMPLTS" in the WRITE statement.

h. omit the END statement.

i. press RETURN twice after END.

The experience you gain in recognizing and handling errors will pay huge dividends later!

Chapter 7:
WRITE AND READ STATEMENTS

A. WRITE STATEMENTS

1. Form

All output in FORTRAN, whether to the console, the printer, or a data file, is governed by the WRITE statement. Here is its form:

```
WRITE (unit number, format statement line number) output list
```

a. Unit number

The unit number specifies the output device; for now we'll use "*", which means "CONSOLE". Later, we will discuss how to send output to the printer or to a data (text) file.

b. Format statement line number

Every time you want output in FORTRAN, you must specify the format you desire for that output. This is done by way of the FORMAT statement (the subject of our next lesson). Every FORMAT statement is preceded by a line number, and this line number is what is included in the WRITE command.

c. Variable(s) and/or message(s) and/or expression(s)

The last thing required by WRITE is an output list. If you list several items, they must be separated by commas. Messages are enclosed in *single quotes* in FORTRAN. Variables and messages may be combined in a single WRITE, again with commas between them. It is also valid to place an expression in a WRITE statement (as you have seen in your first program, where "MILES/GALLNS" is calculated within a WRITE).

2. Examples

```
WRITE (*, 100) FLAG
```

Write the contents of variable FLAG to the Console (*) according to the FORMAT given in line 100.

NOTE: The FORMAT statement line referenced may occur anywhere in

the program (not necessarily before the WRITE statement). Some programmers like to place the WRITE and FORMAT statements together, others will list all FORMAT statements either at the beginning or the end of the program.

```
WRITE (*, 200) COURSE, TEACHR, POINTS / HOURS
```

Write the contents of variables COURSE, TEACHR, and the expression "POINTS/HOURS" to the console according to the FORMAT described in line 200.

```
WRITE (*, 300) 'Type your name: '
```

Write the message, according to the FORMAT statement in line 300, on the console.

B. READ STATEMENTS

1. Form

All input in FORTRAN, whether from the keyboard or from a data file, is governed by the READ statement. Here is its form:

```
READ (unit number, format statement line number) variable(s)
```

Notice that READ and WRITE are identical in form, so there is no need to explain the parameters you must supply.

NOTE: "READ" can only accept data; "WRITE" can only print data. FORTRAN will not allow you to combine input and output into one statement (as will BASIC—INPUT "What is your name? "; NAME$). Whenever you ask the program user a question, you must use a "WRITE" to print the question, and a "READ" to accept the answer.

2. Examples

```
READ (*, 100) HEIGHT
```

Read a value from the keyboard according to FORMAT line 100, and store it in variable HEIGHT.

```
READ (*, 200) LENGTH, WIDTH, DEPTH
```

Input three data fields from the keyboard according to line 200, and store them as LENGTH, WIDTH, and DEPTH.

In a later lesson we will learn how to read from a file instead of the terminal.

C. EXERCISE

Type the PASSES program you entered for the last set of exercises, but omit the variable "PERCNT". (This is possible since calculations may be performed within a WRITE statement.) R(UN the program.

Chapter 8:
FORMAT STATEMENTS

FORMATs serve as a "blueprint" for one line of input or output. They tell "WRITE" precisely how and where to place output, and they tell "READ" precisely how to retrieve input.

All FORMAT statements have this form:

```
line number     FORMAT (format specifier, format specifier, etc.)
```

Recall that line numbers must appear within the first five columns of the program line. The keyword "FORMAT" must begin no earlier than Column 7.

You should know that *every* "READ" and "WRITE" requires an associated FORMAT. Fortunately, *it is possible to specify a format once, then refer to it (via its line number) in more than one READ or WRITE.* This prevents the number of FORMATs needed from getting out of hand.

On to the Format specifiers:

A. NUMERIC OUTPUT AND INPUT FORMATS

1. Integer data

a. Form

```
Iw
```

"w" is an integer specifying Field Width.

NOTE: If you specify an "I" format, all characters in that field must be Integers. If the field contains a letter, a decimal or an exponent, you will receive an error message. In addition, the variable used in the associated "READ" or "WRITE" statement must be an Integer variable. In other words, you could not "WRITE" the contents of the variable "DEGREE" with an "I" Format specifier.

b. Examples

I5 a five column integer field

I1 a one column integer field

c. Integer Format with WRITE

NOTE: The "I" FORMAT is always RIGHT-JUSTIFIED. Study the following tables carefully and be sure you understand them.

Let's print some values with an "I4" FORMAT. The left column represents the values as they are stored, the **emphasized** right-column represents the output:

VALUES	IIII
12	**12**
345	**345**
-2	**-2**
1ØØ4	**1ØØ4**
3	**3**
12345	********

d. Things to note from the above table:

1. Numbers are right-justified (lined up from the right). This means that the integer will be preceded by spaces if it is smaller than the field width.
2. Negative signs take up one field character, just as the ten digits (0–9) do.
3. If a number is too large to fit in the assigned field, a row of asterisks will appear.

e. Integer Format with READ

Let's read some values and store them. The table below shows the results when READing with an "I4" FORMAT. The **emphasized** numbers on the left are what the program user types, the numbers on the right represent how those numbers were read:

IIII	READ AS
1	1ØØØ
162	162
12	12
-4	-4
24	24Ø
49152	4915

f. Things to note from the above table:

1. You must right-justify your input. In other words, if the field is I4, but your number has only two digits, you must precede those two digits with two spaces.
2. If you omit leading spaces, your number will be padded with zeroes.
3. If you over-shoot the field (type five digits in an I4 field, for example), only those digits within the field will be accepted.

2. Floating-Point (Real) Data

a. Form

```
Fw.d
```

"w" is an integer specifying field width, just as it is with Integer FORMAT.

"d" is an integer specifying how many digits will occur *after* the decimal.

b. Examples

```
F7.2
```

Sets a total field width of seven columns (*including* the decimal and negative sign), with two occurring after the decimal. Here are a few F7.2 numbers: 3451.23, 9001.29, and −142.96

```
F6.3
```

The total field width is six columns, three of which are after the decimal. Some F6.3 numbers are: 23.528, −2.419, and 90.013

c. Floating-Point Format with WRITE

Let's print some values with an F6.2 FORMAT. The left column represents the values stored, the **emphasized** right-column the output:

VALUES	FFF.FF
12.4	12.4Ø
-12.Ø1	-12.Ø1
1.Ø39	1.Ø4
21632.2	******
Ø.Ø33	Ø.Ø3
18Ø1.1	******

d. Things to note from the above table:

1. Again, output is right-justified. Leading spaces are supplied when necessary.
2. If the number of digits *after* the decimal is *fewer* than the FORMAT calls for, the number is padded with zeroes.
3. If the number of digits *after* the decimal is *greater* than the format calls for, the number is *rounded off* (not truncated).
4. If the *total* number of digits in the format is exceeded, a row of asterisks will appear.
5. Why did the last value cause the asterisks even though it did not exceed the six-character limit?

e. Floating-Point Format with READ

Let's read some values and store them; the results are summarized in the table below. The **emphasized** left column represents what the program user typed; the right column represents what was actually stored. The FORMAT used is "F5.1".

FFF.F	READ AS
12 .8	12Ø.8
.5	Ø.5
28.5	28.5
3 .1	3ØØ.1
4.	4.Ø
129.45	129.4

FFF.F	READ AS
64145	6414.5
3671	367.1
1 93	1ØØ9.3

FFF.F	READ AS
1.49	1.49
23.6	23.6Ø
7.914	7.914
.16	.16ØØ
5175.	5175.

f. Things to note from the above table:

1. The first group of numbers in the table shows that any blanks in input are interpreted as zeroes, and that any digits beyond the format field width (five digits in the example) are ignored. Notice that this is the same as it was with Integer FORMATs.
2. The second division in the table illustrates that when one uses Floating Point FORMAT, *one need not type any decimal at all.* The operating system is then said to use the *implied decimal position.* In this manner, one can read in a number which is actually larger than the specified field width.
3. The third group of numbers in the table illustrates that the format of the number you type does not have to match the FORMAT specified exactly. Recall that the Floating-Point format is "Fw.d" where "w" is field width, and "d" is places after the decimal. *While the field width specifier is absolute, the decimal-place specifier can be over-ridden.* In the table, the FORMAT was "F5.1" but the following FORMATs were read without error: F5.2, F5.2, F5.3, F5.4, and F5.0. Again notice that in the original "F5.1" format, the "5" could not be over-ridden, but the "1" could be.

NOTES:

1. *Since input must be so rigidly formatted in FORTRAN, you must inform the program user of the FORMAT used every time s/he supplies input.*

2. *To reap the full benefits of FORTRAN, you must understand the Integer and Floating-point FORMATs fully for both "READ" and "WRITE". There are two ways to do so: one, study all four tables in this lesson carefully; and two, enter and use program "FORMAT", which is listed at the end of the chapter.*

B. CHARACTER FORMATS

1. To print a message, enclose it in *single quotes* in a WRITE statement, then use the Format specifier "A" in the accompanying FORMAT statement.

2. Example

In the following sample program segment, note that the lengths of the messages are different, yet Format specifier "A" handles them both. This is because "A" uses *implied field width* (i.e., the field width specified in the "CHARACTER" declaration for variables, or the field width of the text enclosed by single quotes for messages).

Also note that the same FORMAT is referenced twice; there is no need to repeat identical FORMATs.

```
      WRITE (*, 1ØØ) 'Wow!'
1ØØ   FORMAT (A)
      WRITE (*, 1ØØ) 'This sure is a fascinating lesson!'
```

3. "A" also governs input and output of Character variables: again, field width will be implied. Two examples follow:

```
      READ (*, 75) REPLY
75    FORMAT (A)

      WRITE (*, 1ØØ) STREET
1ØØ   FORMAT (A)
```

Of course, REPLY and STREET would have been declared as Character variables at the start of their respective programs.

4. Combining character and numeric output

a. Here are some examples using WRITE:

```
      WRITE (*, 1ØØ) 'The answer is ', MAX
1ØØ   FORMAT (A, I4)

      WRITE (*, 5Ø) 'City of birth: ', CITY
5Ø    FORMAT (A, A)

      WRITE (*, 25Ø) 'You now have ', NUMBER,
     *               ' numbers with average ', AVERGE
25Ø   FORMAT (A, I2, A, F6.3)
```

Note the use of the continuation character "*" in column 6 of the last example. The Compiler will read the first two lines as if they were really one. *You should break a line only at commas.*

Notice that character and numeric Format specifiers are placed in the

same FORMAT statement, with a *comma* separating them. Also note the one-to-one correspondence between the output list in the WRITEs and the Format specifiers in the FORMATs *(Integer variables --> "I", Real variables --> "F"; Character variables and messages --> "A").*

NOTE: NO MATTER HOW MANY SPECIFIERS ARE LISTED IN A FORMAT STATEMENT, IT WILL ALWAYS PRODUCE EXACTLY ONE LINE OF OUTPUT. The end of the FORMAT statement will then generate the "RETURN" character, ASCII 13 (thereby causing the cursor to move down to the next line, just as if the "RETURN" key was pressed).

b. Now let's use READ and WRITE to answer a question:

```
        WRITE (*, 1Ø) 'Your choice? '
1Ø      FORMAT (A)
        READ (*, 2Ø) ITEM
2Ø      FORMAT (I1)
```

These four lines work, but the prompt and the answer will be on *separate* lines!

c. To get both the question and the answer on the *same* line, use the special Format specifier "$" after "A" (*this has absolutely nothing to do with the "string" designation of BASIC*). Think of "$" as being an "S", as in *"$tay put, cursor!"* or *"$uppress RETURN"*. Let's do the above example again:

```
        WRITE (*, 1Ø) 'Your choice? '
1Ø      FORMAT (A, $)
        READ (*, 2Ø) ITEM
2Ø      FORMAT (I1)
```

5. There will be times when a programmer would like a *specific* number of characters reserved in his/her output (for example, when producing a table) or in input (especially when reading from data files). This is possible by using the following form of the "A" FORMAT:

```
Aw
```

where "w" is a field width specifier.

a. "Aw" Format with WRITE
Shown below is a table with the results of printing strings of different lengths with the Format specifier "A6". The table con-

sists of the CHARACTER declaration for the variable, the strings as they are stored, and the **emphasized** actual output; "Ø" stands for "space."

DECLARATION	VALUE STORED	AAAAAA
PLANET*1Ø	EARTHØØØØØ	**EARTHØ**
PLANET*1Ø	JUPITERØØØ	**JUPITE**
PLANET*4	MOON	**ØØMOON**
PLANET*4	XØØØ	**ØØXØØØ**

b. Things to note from the above table:

1. A field width *less* than the CHARACTER declaration allows *only those characters that will fit within the field to be printed.* This has the effect of *left-justifying* as many characters as the field width permits.
2. A field width *larger* than the CHARACTER declaration causes the string to be *preceded by spaces* (to make up the difference). This has the effect of *right-justifying* the characters as stored.
3. *Since tables usually have non-numeric entries left-justified, it is usually wiser to select a field width that is LESS THAN the CHARACTER declaration. In fact, the wisest choice is to use the same field width as your declaration (in which case you may use implied field width).*

c. "Aw" Format with READ

The following table shows the results of reading strings of varying lengths with an "A6" FORMAT. The **emphasized** strings on the left are those typed, the strings on the right represent how the operating system interprets them.

AAAAAA	READ AS
BETTY	BETTYØ
JOAN	JOANØØ
GEORGETTE	GEORGE

d. Things to note from the above table:

1. The operating system will add trailing blanks if the input field is too small.

2. The operating system will *ignore* any characters beyond the specified field width.

C. POSITIONAL FORMAT SPECIFIERS

1. To skip spaces, one must use the following Format specifier:

```
nX
```

where "n" is an integer indicating how many spaces to skip.

NOTE: Even a single space skip MUST be prefaced by an integer ("1X", not "X").

2. Examples and explanations

```
5X      skip five spaces from the present cursor position

23X     skip twenty-three spaces
```

3. Unfortunately, tabbing is not possible in Apple FORTRAN, but we will briefly discuss its use anyway. The form is:

```
Tn
```

where "n" is an integer indicating the column to tab to.

4. Examples and explanations

```
T2Ø     tab to column 20 unless you're already past it

T7Ø     tab to column 70
```

Note that "X" is a *relative* FORMAT (go from where the cursor is), while "T" is an *absolute* FORMAT (go there no matter what your position).

D. GOVERNING MORE THAN ONE LINE OF INPUT OR OUTPUT

So far, the rule has been that *one FORMAT statement governs one line of input or output.* How can we print several lines with only *one* WRITE and *one* FORMAT (rather than one of each for every line)? The answer is by using the character "/".

"/" can be interpreted as meaning "end of the current input or output

line." What it actually does is generate the RETURN character (ASCII code 13).

Here is an example that prints two lines:

```
        WRITE (*, 1ØØ) 'Better luck', 'next year'
1ØØ     FORMAT (A, /, A)
```

Now let's skip a line between the messages:

```
        WRITE (*, 1ØØ) 'Better luck', 'next year'
1ØØ     FORMAT (A, /, /, A)
```

Why was only one line skipped (not two)?

Finally, an example of "/" with READ:

```
        READ (*, 1ØØ) NUMBER, PRICE
1ØØ     FORMAT (I3, /, F5.3)
```

This will read a 3-digit Integer from one line, then an F5.3 Real number from the next line (you will still need to press RETURN twice, once for each line).

E. SUMMARY

The following table summarizes the Format specifiers available in FORTRAN.

FORM	EXAMPLES	SPECIFIES
Iw	I5	INTEGERS
Fw.d	F4.1	REAL NUMBERS
A	A	CHARACTERS (field width is implied)
Aw	A1Ø	CHARACTERS (specified field width)
$	$	STAY ON SAME LINE ($uppress RETURN)
nX	5X	SPACES
Tn	T2Ø	TABS (not available in Apple FORTRAN)
/	/	ADVANCE TO NEXT LINE (generate RETURN)

One FORMAT statement may contain any or all of the above specifiers. Unless it contains the slash (/), it will govern only one line of input or output.

F. REVIEW QUESTIONS

1. Show the precise output for these ten program segments (use "Ø" for all blanks):

a.
```
      WRITE (*, 1Ø) -46.837
1Ø    FORMAT (F6.1)
```

b.
```
      WRITE (*, 1Ø) 128.3
1Ø    FORMAT (F7.3)
```

c.
```
      WRITE (*, 1Ø) 48.Ø1
1Ø    FORMAT (F4.2)
```

d.
```
      WRITE (*, 1Ø) 28
1Ø    FORMAT (I3)
```

e.
```
      WRITE (*, 1Ø) 6
1Ø    FORMAT (I4)
```

f.
```
      WRITE (*, 1Ø) 237
1Ø    FORMAT (I2)
```

g.
```
      WRITE (*, 1Ø) 'Triangle'
1Ø    FORMAT (A)
```

h.
```
      WRITE (*, 1Ø) 'Rectangle'
1Ø    FORMAT (A1Ø)
```

i.
```
      WRITE (*, 1Ø) 'Pentagon'
1Ø    FORMAT (A5)
```

j.
```
      WRITE (*, 1Ø) 'HexagonØØ'
1Ø    FORMAT (A12)
```

2. Now find the exact value stored when the program user types the values listed, which are then READ with the given FORMATs:

	VALUE AS TYPED	FORMAT USED TO "READ"
a.	Ø3	I3
b.	6	I4
c.	Ø28	I2
d.	Ø2.6	F4.3
e.	286	F3.1
f.	249.6	F4.1
g.	Single	A
h.	Double	A1Ø
i.	Triple	A1

3. A programmer types the following program segment and the Compiler gives him/her an error message. Can you find the mistake?

```
        WRITE (*, 5Ø) TEMP, PRECIP
5Ø      FORMAT (I3, F5.2)
```

G. EXERCISE

Type the following program and execute it until you know your Format specifiers inside and out. It will allow you to test your skill with the Format specifier of your choice, or allow you to combine all the specifiers into one large 'FORMAT' statement which will govern the printout of an address label for an envelope. (*NOTE: Don't be concerned with deciphering the entire source program, for it incorporates many features of FORTRAN which we have yet to discuss. Simply concentrate on executing it.*)

Note on the "Address label" option: all the program user need do is supply a list of Format specifiers in parentheses (use neither a line number nor the word "FORMAT"); don't forget the correspondence between variable types and their Format specifiers (so you *must* use "A", "I", and "F"). You will also be able to use the "nX" and "/" specifiers if you wish. Although you may get a barrage of Run-time errors your first few attempts with this option, with practice you should become quite proficient at supplying valid Format specifiers!

```
        PROGRAM FORMAT

C       Type declarations
        CHARACTER HOME*1, INVRSE*1, NORMAL*1, BELL*1, DELAY*1
        INTEGER OPTION, RINGS

C       Special character definitions
        HOME = CHAR (12)
        INVRSE = CHAR (15)
        NORMAL = CHAR (14)
        BELL = CHAR (7)
        ASSIGN 3Ø TO MENU

C       Title page
        WRITE (*, 5) HOME
5       FORMAT (A, $)
        WRITE (*, 1Ø) 'Welcome to'
1Ø      FORMAT (/, /, /, 19X, A, /)
        WRITE (*, 2Ø)
     *     INVRSE, '*** LEARNING APPLE FORTRAN FORMATS ***', NORMAL
2Ø      FORMAT (5X, A, A, A)
```

```
3Ø      WRITE (*, 35) 'Press RETURN to continue...'
35      FORMAT (/, /, /, 1ØX, A, $)
        READ (*, 4Ø) DELAY
4Ø      FORMAT (A)

C       Menu Page
45      WRITE (*, 5) HOME
        WRITE (*, 5Ø) INVRSE, 'Available options', NORMAL
5Ø      FORMAT (1ØX, A, A, A, /)

        WRITE (*, 6Ø) '1. Integer (I) Format'
6Ø      FORMAT (7X, A)
        WRITE (*, 6Ø) '2. Floating point (F) Format'
        WRITE (*, 6Ø) '3. Character (A) Format'
        WRITE (*, 6Ø) '4. Exponential (E) Format'
        WRITE (*, 6Ø) '5. Format an address label'
        WRITE (*, 6Ø) '6. Quit'

        WRITE (*, 7Ø) 'Your choice (1 - 6) ? '
7Ø      FORMAT (/, /, 1ØX, A, $)
        READ (*, 75) OPTION
75      FORMAT (I1)

        WRITE (*, 5) HOME
        GOTO (9Ø, 1ØØ, 11Ø, 12Ø, 13Ø, 14Ø) OPTION

C       Otherwise invalid reply

        DO 8Ø RINGS = 1, 5
           WRITE (*, 5) BELL
8Ø      CONTINUE

        WRITE (*, 4Ø) 'Invalid option!'
        GOTO MENU

9Ø      CALL IFORMT
        GOTO MENU

1ØØ     CALL FFORMT
        GOTO MENU

11Ø     CALL AFORMT
        GOTO MENU
```

```
12Ø     CALL EFORMT
        GOTO MENU

13Ø     CALL ADDRSS
        GOTO MENU

14Ø     END


        SUBROUTINE IFORMT
C       Integer Format option

        WRITE (*, 1Ø) 'IIII'
1Ø      FORMAT (28X, A)
        WRITE (*, 2Ø) 'Type an I4 format Integer : '
2Ø      FORMAT (A, $)
        READ (*, 3Ø) NUMBER
3Ø      FORMAT (I4)
        WRITE (*, 4Ø) 'Your number was read as ', NUMBER
4Ø      FORMAT (/, A, I4)

        END


        SUBROUTINE FFORMT
C       Floating point Format option

        WRITE (*, 1Ø) 'FFF.FF'
1Ø      FORMAT (34X, A)
        WRITE (*, 2Ø) 'Type an F6.2 format Real number : '
2Ø      FORMAT (A, $)
        READ (*, 3Ø) ANSWER
3Ø      FORMAT (F6.2)
        WRITE (*, 4Ø) 'Your answer was read as ', ANSWER
4Ø      FORMAT (/, A, F11.5)

        END


        SUBROUTINE AFORMT
C       Character Format option

        CHARACTER CHARS*1Ø
```

```
      WRITE (*, 1Ø) 'AAAAAAAAAA'
1Ø    FORMAT (38X, A)
      WRITE (*, 2Ø) 'Type an A1Ø format character string : '
2Ø    FORMAT (A, $)
      READ (*, 3Ø) CHARS
3Ø    FORMAT (A1Ø)
      WRITE (*, 4Ø) 'Your answer was read as ', CHARS
4Ø    FORMAT (/, A, A1Ø)

      END


      SUBROUTINE EFORMT
C     Exponential Format option

      WRITE (*, 1Ø) 'Note : "FLOATING POINT ERROR" means your'
1Ø    FORMAT (A)
      WRITE (*, 1Ø) 'number was either larger than 1E+38 or'
      WRITE (*, 1Ø) 'smaller than 1E-38 (APPLE FORTRAN''s limit)'

      WRITE (*, 2Ø) '.DDEEE'
2Ø    FORMAT (/, /, 41X, A)
      WRITE (*, 3Ø) 'Type an E6.2 format exponential number : '
3Ø    FORMAT (A, $)
      READ (*, 4Ø) ANSWER
4Ø    FORMAT (E6.2)
      WRITE (*, 5Ø) 'Your answer was read as ', ANSWER
5Ø    FORMAT (/, A, E12.6)

      END

      SUBROUTINE ADDRSS
C     Address label option

C     Type declarations
      CHARACTER NAME*3Ø, STREET*3Ø, CITY*25, COMMA*1
      CHARACTER STATE*2Ø, LABEL*75, HOME*1
      INTEGER STRTNO

C     Data for address label variables
      DATA NAME, STRTNO / 'Premontre High School', 61Ø /
      DATA STREET, COMMA, CITY / 'Maryhill Drive', ',', 'Green Bay'/
      DATA STATE, ZIP / 'Wisconsin', 543Ø3.Ø /

      HOME = CHAR (12)
```

```
C       Directions
        WRITE (*, 1Ø) 'You are to enter a SINGLE Format statement to'
1Ø      FORMAT (A)
        WRITE (*, 1Ø) 'address an envelope with these given variables'
        WRITE (*, 1Ø) 'and their corresponding values :'

C       Report Values
        WRITE (*, 2Ø) 'NAME (Character*3Ø)        : ', NAME
2Ø      FORMAT (/, /, A, A)
        WRITE (*, 3Ø) 'STREET NUMBER (Integer)    : ', STRTNO
3Ø      FORMAT (A, I5)
        WRITE (*, 4Ø) 'STREET NAME (Character*3Ø) : ', STREET
4Ø      FORMAT (A, A)
        WRITE (*, 4Ø) 'CITY (Character*25)        : ', CITY
        WRITE (*, 4Ø) 'COMMA (Character*1)        : ', COMMA
        WRITE (*, 4Ø) 'STATE (Character*20)       : ', STATE
        WRITE (*, 5Ø) 'ZIP CODE (Real)            : ', ZIP
5Ø      FORMAT (A, F6.Ø)

C       Accept label Format
        WRITE (*, 6Ø)
     *       'Type a SINGLE Format statement by the ">" below'
6Ø      FORMAT (/, /, A)
        WRITE (*, 7Ø) 'NOTE 1 : DON''T supply a line number!'
7Ø      FORMAT (3X, A)
        WRITE (*, 7Ø)
     *       'NOTE 2 : DON''T type the keyword FORMAT!'
        WRITE (*, 8Ø)
     *       'NOTE 3 : DO enclose Format specifiers in parentheses!'
8Ø      FORMAT (3X, A, /, /)
        WRITE (*, 9Ø) '> '
9Ø      FORMAT (A, $)
        READ (*, 1Ø) LABEL

C       Print label
        WRITE (*, 9Ø) HOME
        WRITE (*, 4Ø) 'Format for label : ', LABEL
        WRITE (*, LABEL) NAME, STRTNO, STREET, CITY, COMMA, STATE, ZIP

        END
```

Chapter 9:
TRANSFER OF CONTROL AND CONDITIONALS

A. TRANSFER OF CONTROL

1. GOTO line number

This is used to branch unconditionally to a different part of the program. You now know two FORTRAN instruction lines which require line numbers: FORMAT statements and those lines which are targets of GOTO branching.

Examples:

```
GOTO 1ØØ
```

```
GOTO 25Ø
```

2. Computed GOTO

This form of branching is commonly used to avoid a series of tests after giving the program user a menu option. In other words, the target of the branching depends on the value of a variable (and that value is usually typed by the person executing the program).

a. Form

```
GOTO (line number, line number, line number, . . .) Integer variable
```

b. Examples and Explanations

```
GOTO (1ØØ, 2ØØ, 3ØØ) ITEM
```

Branching will be to line 100 if ITEM = 1, 200 if ITEM = 2, or 300 if ITEM = 3.

```
GOTO (9ØØ, 2ØØ) ITEM
```

```
GOTO (5, 8ØØ, 2Ø, 3ØØ, 1ØØ) NUMBER
```

c. Notes:

1. The variable listed *must* be an Integer variable (actually, it may be an Integer *expression,* also).
2. You may have as many line numbers in parentheses as you wish.
3. If the variable listed has a value either less than 1 or greater than the number of line numbers you have listed, the statement will be *ignored*. This allows a programmer to use error-trapping in the lines immediately below the Computed GOTO.
4. If you would like to see an example of the Computed GOTO used within the context of a program (complete with error-trapping), refer to the "FORMAT" program given at the end of the last chapter.

B. CONDITIONALS

1. Arithmetic IF

This form of decision is used when the sign of an expression (negative, equal to 0, or positive) affects which instructions must be followed.

a. Form

```
IF (expression) line number, line number, line number
```

b. Sample program segment

Note how branching will be to line 100 if SLOPE is negative, 110 if SLOPE is zero, and 120 if SLOPE is positive.

```
        IF (SLOPE) 1ØØ, 11Ø, 12Ø

1ØØ     WRITE (*, 1Ø5) 'Negative slope'
1Ø5     FORMAT (A)
        GOTO 13Ø

11Ø     WRITE (*, 1Ø5) 'Horizontal (Ø slope)'
        GOTO 13Ø
```

```
12Ø     WRITE (*, 1Ø5) 'Positive slope'

13Ø     program continues. . .
```

c. Note: With the Arithmetic IF one *must* supply three line numbers, no more and no fewer.

2. Logical (Single-instruction) IF

This conditional structure is used when a *single* instruction will be executed only if a decision is true (i.e., a single **Yes**-branch instruction).

a. Form

```
IF (expression .logical operator. expression) instruction
```

where "logical operator" is one of the following:

GT	greater than
LT	less than
EQ	equal to
NE	not equal to
GE	greater than or equal to
LE	less than or equal to

"Instruction" is any valid FORTRAN statement except DO (a looping command which we will learn 2 chapters hence) or another IF.

b. Examples

```
IF (SCORE .GT. 9Ø.Ø) BONUS = SCORE * Ø.1Ø

IF (YEARS .LT. 2.5) GOTO 2Ø

IF (AVE .GT. 93.Ø) WRITE (*, 1ØØ) 'A-'

IF (A**3 + B**3 .LE. Ø) READ (*, 1ØØ) A, B
```

NOTES:

1. *There is no "THEN" following the "IF" in this form of the conditional.*

2. *When you are comparing numeric values, it is not necessary to worry about numeric types (Integer and Real). If the types don't match, the Integer expression is automatically converted to Real (thereby insuring matching types).*

c. END is ABSOLUTE

You should recall that the "END" statement is absolute. It literally shuts the compiler down! Therefore, the *only* place you may use "END" is as the last line of your program. The following two program segments are *invalid:*

```
        IF (REPLY .EQ. 'Y') END

        IF (ANSWER .NE. 1) GOTO 1Ø
        END
1Ø      WRITE (*, 2Ø) RESULT
```

Again, these two segments are invalid since "END" is *not* the last statement of the program. The Compiler will shut down when it sees "END", plus give you some error messages for tacking on extra lines. The solution to this problem is to assign the END statement a line number, then use a GOTO in its place.

3. Block (Several instruction) IF. . .THEN. . .ELSE

a. This is a very useful *compound* (multiple-line) structure for coding decisions with several Yes- and No-branch instructions. It allows a very readable, GOTO-less conditional, and has two forms:

```
IF (expression .logical operator. expression) THEN
   yes branch instructions
   :
   :
   :
ENDIF
```

The above structure is used if there are Yes-branch instructions but no No-branch instructions. The following is used if there are both Yes- and No-branch instructions:

```
IF (expression .logical operator. expression) THEN
   yes branch instructions
   :
   :
ELSE
   no branch instructions

   :
ENDIF
```

b. Examples

```
IF (EARNED .GE. 2ØØ.ØØ) THEN
   EXMPTN = 21.55
   PERCNT = Ø.78
ENDIF

IF (SEX .EQ. 'M') THEN
   WRITE (*, 1Ø) 'You sure are a handsome young man.'
   WRITE (*, 1Ø) 'What is that after-shave you''re wearing?'
ELSE
   WRITE (*, 1Ø) 'You sure are a pretty young lady.'
   WRITE (*, 1Ø) 'What is that perfume you''re wearing?'
ENDIF
```

NOTES:

1. *It is not possible to put more than one statement per line in a FORTRAN program. If your decision branches have several instructions, you must use the Block "IF. . .THEN. . .ELSE" structure.*

2. *There may be no instruction on the same line as the word "THEN" in this structure.*

3. *Note how the indentation of the Yes- and No-branch instructions in the above examples increases program readability!*

4. Testing Character variables

a. Checking one Character variable against another. Example:

```
IF (REPLY .EQ. CITY) GOTO 3ØØ
```

REPLY and CITY would have been declared as Character variables at the start of the program.

b. Checking Character variables' contents. This form was hinted at in one of the examples above. Example:

```
IF (ANSWER .NE. 'END') GOTO 25Ø
```

Notice the single quotes.

c. Alphabetizing. This is done exactly as it is in BASIC: one string is "less than" another if it is *earlier* in alphabetical order (an easy way to remember this is to think of a "smaller" string as being on a smaller page number if it were looked up in a dictionary or phone book). Example:

```
IF (NAME1 .LT. NAME2) WRITE (*, 5ØØ) NAME1
```

d. It is illegal to compare a Character variable with a numeric (Integer or Real) variable or value.

C. REVIEW QUESTIONS

1. What would happen if a programmer listed a menu with five options, but listed only four line numbers in her/his Computed GOTO statement?

2. Why do you think the Logical (one-instruction) IF has no "THEN", but the Block (several-instruction) IF does?

3. When does the Block IF have no "ELSE" statement?

4. What do you think would happen if you were to compare two Character variables of unequal length? Would an error message be reported?

D. EXERCISE

Write a program to determine the Federal Income Tax withholding for a program user-supplied salary. For simplicity, we will divide earnings into only three tax brackets as follows:

CUTOFF	STD DEDUCTION	PLUS X%	OF AMOUNT OVER
$1Ø,ØØØ	$ Ø	7%	$ 2,5ØØ
$2Ø,ØØØ	$ 6ØØ	1Ø%	$12,5ØØ
$2Ø,ØØ1+	$15ØØ	15%	$22,5ØØ

The program should ask for a person's annual salary, then produce output similar to the following:

```
Enter your earnings for the year: $25000.00

Standard deduction: $1500.00
+ 15% of $2500.00 : $ 375.00
                    -------
Total income tax  : $1875.00

Would you like another run (Y/N) ? N
```

You will surely learn to appreciate FORTRAN's ability to position numbers in columns, as well as round figures to a specific number of decimal positions, during the course of this program!

Chapter 10:
ODDS AND ENDS

A. COMMENT LINES

There are two ways to produce a Comment (i.e., non-executed) line.

1. Place the letter "C" in Column one

```
C       These are comment lines.
C       Notice the ''C'' in Column one.
C       The Compiler will ignore these lines.
```

Comment lines may *not* be continued with an asterisk.

2. Leaving an entire line blank

Blank lines are generated by pressing the RETURN key *twice* at the end of a line. They greatly increase program readability, and have no effect on the Compiler. In a well-organized program, each subunit (such as a loop, IF. . .THEN. . .ELSE, subroutine, etc.) can be easily set off from the rest of the program by blank lines (and/or indentation). We've been doing so all along!

B. ADDENDUM TO CONDITIONALS

FORTRAN supports the logical operators AND and OR in conditionals. Study the examples below:

```
          IF (REPLY .LT. 'A'  .OR.  REPLY .GT. 'E') THEN
C            Chastise and return to menu
             WRITE (*, 10) 'Invalid reply!'
             GOTO 50
          ENDIF

          IF (FIRST .NE. SECOND  .AND.  FIRST .NE. THIRD) GOTO 80
```

Like all logical operators, AND and OR require periods before and after their use. To enhance readability, it is strongly recommended that you use *two* spaces before and after AND and OR. (I hope you noticed that we have been using *one* space before and after the *other* logical operators. If not, shame on you!)

Be sure that decisions joined with AND and OR are *complete* decisions. The following examples are *incorrect:*

```
IF (NAME .EQ. 'ALPHONSE'  .OR.  'ZELMO') GOTO 5Ø

IF (ITEM .GT. Ø  .AND.  .LT. 5) GOTO 1ØØ
```

AND and OR may be used in the Logical and Block conditional structure.

C. CHANGING DEFAULT VARIABLE TYPES

If at all possible, one should use *descriptive variable names* in a program. A descriptive variable is one whose purpose can be implied from its name. The variables "E6", "Q", "M2", and "W" give us no clue as to what they store, but "REPLY", "DATE", "GRADE", and "TAX" do! Well-chosen variable names make a program easier to read and to debug. (Aside: It is not *always* possible to dream up a descriptive variable; sometimes "X" is simply "X".)

FORTRAN's default variable types pose a bit of a problem, however. Variable "MONEY" can only store Integer values, but suppose you want it to store Real values. You could use "AMONEY" to satisfy the default, but that name looks a bit strange. If only you could override the default, descriptive variables would be so easy to come by. . .

1. The REAL Statement

You should recall that variables beginning with the letters A–H and O–Z are assumed to be Real. What if you have your heart set on using "NUMBER" as a Real variable? The "REAL" statement will come to your rescue! Here is its form:

```
REAL list of variables you want to be real
```

Here are some sample uses:

```
REAL MCCOY
REAL NEAT
REAL IDIOT, MAX, LIMIT
```

In the above examples, the variables defined as Reals are normally Integers. This statement *overrides* the default.

2. The INTEGER Statement

Guess what this does? Right, it declares variables that would normally be Real as Integer variables instead. A few samples follow:

```
INTEGER VALUE
INTEGER ANSWER
INTEGER REPLY, DELAY, SPEED
```

As with the "REAL" statement, "INTEGER" may name more than one variable as long as each variable is separated from the others by a comma.

3. The IMPLICIT Statement

What would happen if you were writing a huge program and needed to declare 20 variables as Integers? Would you have to list all 20 in an INTEGER statement? Fortunately, no.

FORTRAN allows one to change the default for determining variable types with the IMPLICIT statement. IMPLICIT has one of three forms:

```
IMPLICIT INTEGER (starting letter - ending letter)

IMPLICIT REAL (starting letter - ending letter)

IMPLICIT CHARACTER*length (starting letter - ending letter)
```

Here are some samples and explanations. *Keep in mind that IMPLICIT overrides only the range of letters specified.*

```
IMPLICIT INTEGER (A - H)
```

Now all variables beginning with letters A–N are Integers (recall that I–N were *already* Integers).

```
IMPLICIT REAL (I - N)
```

Now *all* variables are Real.

```
IMPLICIT CHARACTER*20 (B - D)
```

Variables beginning with B, C, or D are now Character variables, each with a 20-byte length.

```
IMPLICIT INTEGER (A - C, W - Z)
```

It is possible to specify two or more ranges in an IMPLICIT. Just separate the ranges with commas.

```
IMPLICIT REAL (J)
```

You may also pick just a single letter for your IMPLICIT.

```
IMPLICIT CHARACTER*1Ø (A, E - G, Z)
```

Can you interpret the above statement?

NOTES:

1. *"IMPLICIT" affects all variables beginning with the specified letters. "INTEGER" and "REAL" affect only the specific variables that are named. Beware of this important distinction.*

2. *"IMPLICIT" must be the second line of a program if used. The Type statements (INTEGER, REAL, and CHARACTER) must come immediately after "IMPLICIT". This table summarizes the ordering of statements our Compiler enforces:*

STATEMENT	COMMENTS
PROGRAM	*always* first
IMPLICIT	optional, second if present
INTEGER, REAL, or CHARACTER	any order you wish
DATA	fourth on the "ladder"
<rest of program>	
:	
END	*always* last

D. THE "E" FORMAT SPECIFIER

So far we've learned how to read and write Integer, Real, and Character values. Very large values (or very small decimals) may be entered and stored by Real variables in FORTRAN by using Exponential (or Scientific) Notation.

A special Format code is used for reading and writing this special kind of value. Here is its form:

```
Ew.d
```

"w" refers to total field width

NOTE: The field width should be chosen as 1 (for a negative sign, should it appear) + *1* (for the decimal point, which always comes before any digits) + *d* (the number of decimal places you want displayed) + *4* (the special character "E", followed by a plus or minus, followed by a two-digit exponent).

"d" refers to the number of digits after the decimal.

A quick way to determine the correct "E" *format specifier is to choose the number of decimal digits you want to appear, then add 6 to find the proper field width.* For example, should you want 3 decimal digits, use field width 3 + 6 = 9 (FORMAT E9.3).

1. E format with WRITE

This new FORMAT will be easier to understand if we do some sample WRITEs. In the following table, the left column contains the stored values, the **emphasized** right column shows the output of those values with the format "E9.3":

VALUES	S.DDDEEEE
3145.Ø	.315E+Ø4
-25.Ø	-.25ØE+Ø2
193423.Ø	.193E+Ø6
Ø.2849	.285E+ØØ
Ø.ØØ19	.19ØE-Ø2

In the above table, the first character in the FORMAT field is either a negative sign, or *a blank if the number is positive.* (The "S" in the right column's heading stands for "sign".) The decimal is placed before the digits (as will always happen with the "E" format), then there are three D's (corresponding to the three decimal places requested), and finally four E's (the four positions taken by the "E" character, its sign, and its two-digit exponent).

The "E" in the output stands for "* 1Ø**", or "times 10 raised to the power. . ." The largest absolute value that may be stored by Apple FORTRAN is 1E+38, while the smallest non-zero absolute value is 1E-38.

What would happen if you tried to print the above values with an "E7.3" FORMAT? You would receive seven asterisks for each number. The field width of 7 doesn't leave enough room for the 1 (for a possible negative sign) + 1 (decimal) + 3 (decimal places) + 4 (exponent positions) = 9 spaces necessary.

How about an "E8.3" FORMAT? The positive numbers would be printed correctly (this time with *no leading blank*), but the negative number would result in a row of eight asterisks.

How about printing them with an "E10.3" FORMAT? The result would be the same as in the above table except with a leading blank *before* each of the numbers shown. Likewise, an "E11.3" FORMAT would yield two leading blanks plus the results shown in the table.

In other words, it is not possible to print any digits before the decimal. If you specify a field width which is larger than necessary, the extra characters will result in leading blanks (not in digits before the decimal).

2. E format with READ

Again, a table is perhaps the best way to learn how the E FORMAT operates with READ.

NOTE: You shall see that with "READ" it is not necessary to reserve the extra six characters plus the digits after the decimal. In other words, "E7.3" is valid for use with "READ", even though it would not work with "WRITE". Read on to see why. . .

The **emphasized** leftmost column represents the user's typed line, the second column represents the values when read with an E8.3 FORMAT, while the third column shows how the answers would look if they were then printed with an E9.3 FORMAT.

.DDDEEEE	READ AS	PRINTED AS	COMMENTS
.235E+Ø5	.235E+Ø5	.235E+Ø5	
.345E-Ø6	.345E-Ø6	.345E-Ø6	
.1Ø8E Ø2	.1Ø8E+Ø2	.1Ø8E+Ø2	the + is optional
.543E- 3	.543E-Ø3	.543E-Ø3	the zeroes are optional
.2Ø4E 2	.2Ø4E+2Ø	.2Ø4E+2Ø	blanks after E read as Ø
.165E3	.165E+3ØØ	error	too large to be stored
-.825EØ5	-.825E+Ø5	-.825E+Ø5	"-" counts as one char.
-.341E 1	-.341E+Ø1	-.341E+Ø1	

.DDDEEEE	READ AS	PRINTED AS	COMMENTS
123.45E5	123.45E+Ø5	.123E+Ø8	override decimal specifier
1.6 E-5	1.6ØØE-Ø5	.16ØE-Ø4	
198.3E2	198.3E+2Ø	.198E+23	
2.3E-4	2.3E-4ØØ	error	too small to be stored
-3.44E 3	-3.44E+Ø3	-.344E+Ø4	
-143.E-9	-143.E-Ø9	-.143E-Ø6	
185454E9	185.454E+Ø9	.185E+12	implied decimal position
12393E 4	12.393E+Ø4	.124E+Ø6	
-12E- 2	-.Ø12E-2Ø	-.Ø12E-2Ø	
11111111	11111.111	.111E+Ø5	no need to even type an E
18424526	18424.526	.184E+Ø5	implied decimal position
-234.2	-234.2	-.234E+Ø3	override, don't use E

Notice that it is not *essential* to type an "E" when using the "E" FORMAT. In the event that you don't, the "E" FORMAT reverts to an "F" FORMAT. If you anticipate the possibility of an "E", however, you *must* use it. *The "F" FORMAT will not accept the special character "E".*

Study the above table carefully!

E. REVIEW QUESTIONS

1. Do you think the Compiler would allow a conditional such as this:

```
IF (1. .GT. X  .AND.  .5 .LT. X) etc.
```

or would the adjacent decimal points and periods be misinterpreted?

2. Do you think a type declaration such as "INTEGER NUMBER" would cause an error, since **NUMBER** is an Integer variable by default anyway?

3. Given the following type declarations, list the type (Integer, Real, or Character) of each of the variables listed below:

```
IMPLICIT INTEGER (A - C, F)
IMPLICIT CHARACTER*1Ø (M)
INTEGER DOG
REAL LOSER
CHARACTER APE*5
```

a. MAN
b. APE
c. REPLY
d. FIND
e. ARC
f. LOSER
g. DOGS
h. COUNT
i. ON
j. LOSS
k. GAIN
l. PROFIT

F. EXERCISES

1. There is perhaps only one way to learn the intricacies of the "E" FORMAT, and that is by practicing with the "FORMAT" program given at the end of the last chapter. A good 15-minute session would really be worthwhile at this point. If you become bored, you may always try your hand at one of the other FORMAT options. You could even try the Address Label option with an "E" Format specifier for the Zip Code variable. (Can you imagine the look on the Postmaster's face when he sees *that* Zip Code?)

2. One of the most famous pair of "buzzwords" used by computer programmers is *structured programming*. Advocates of Structured Programming techniques (including myself) emphasize that since humans write and debug programs, these programs should be written in a human-readable form (as opposed to an illogical, nearly random collection of cryptic symbols). A well-designed program makes future reference and/or modification much easier.

Here are some of the basic tenets of Structured Programming:

a. Use descriptive variable names.

b. Use blank lines and indentation to block off your program into self-contained, natural subunits (such as IF . . . THEN . . . ELSE conditionals, output sections of a program, groups of related calculations, loops, etc.). These subunits should be introduced with English comment statements.

c. Avoid GOTOs at all costs. Program control should flow naturally and sequentially, rather than jump around haphazardly.

d. Use any other methods you may have at your disposal to make the program more easily readable. (Use blank spaces to separate items in lists, such as format specifiers. Blank spaces should also precede and follow all operators such as "=", "+", "*", and ".GT.". Use mixed upper and lower case within the program, etc.)

Below is a program written by a team of programmers at Nostructure, Incorporated. It produces a wage statement for an employee in the following output format:

```
                HOURS      RATE        PAY
REGULAR          4Ø       $ 5.ØØ    $ 2ØØ.ØØ
OVERTIME          5       $ 7.5Ø    $  37.5Ø

                          TOTAL PAY $ 237.5Ø
```

Although the program works flawlessly, it is all but undecipherable. The program was typed by Joe Scrunch. Mr. Scrunch is in love with the Filer's "Krunch" command, and has adopted it as a philosophy of programming. He has therefore disconnected his computer's spacebar so that his programs may be as compact as possible. Joe would not dream of using blank lines and comments in a program ("What a waste of space!"), nor of using upper- and lower-case text ("It takes too much time and effort to keep clicking the CAPS LOCK key in and out!") The variables for the program were named by Joe Morse, a CIA cryptographer. The decision block in the center of the program was authored by Joe Goto, who loves solving circular mazes. The three stooges at Nostructure, Incorporated, are the kind of people who give FORTRAN a bad name. ("FORTRAN is only suitable for advanced mathematicians and engineers," or "It's a non-expressive and poorly structured language") Now whether or not you have realized it, I have been using Structured Programming techniques from page one of this book. Let's see how much of it has rubbed off on you (either consciously or subconsciously). Rewrite this program so that it sparkles! Strike a blow for FORTRAN's readability and structure! Make me proud of you!

```
      PROGRAM POOR
      INTEGER H,RH,OH
      CHARACTER A*1
      WRITE(*,1Ø)'>>> PRODUCES PAYCHECK REPORT <<<'
1Ø    FORMAT(/,A,/,/)
2Ø    WRITE(*,3Ø)'RATE PER HOUR (F5.2) ? $'
3Ø    FORMAT(/,A,$)
      READ(*,4Ø)R
4Ø    FORMAT(F5.2)
      WRITE(*,3Ø)'HOURS WORKED THIS WEEK (I2) ? '
      READ(*,5Ø)H
5Ø    FORMAT(I2)
      OR=R*1.5
      IF(H.GT.4Ø)GOTO 6Ø
```

```
        RH=H
        OH=Ø
        GOTO 7Ø
6Ø      RH=4Ø
        OH=H-4Ø
7Ø      RP=RH*R
        OP=OH*OR
        WRITE(*,8Ø)'HOURS','RATE','PAY'
8Ø      FORMAT (/,/,/,15X,A,3X,A,8X,A)
        WRITE(*,9Ø)'REGULAR ',RH,'$',R,'$',RP
9Ø      FORMAT(A,8X,I2,4X,A,F5.2,4X,A,F7.2)
        WRITE(*,9Ø)'OVERTIME',OH,'$',OR,'$',OP
        WRITE(*,1ØØ)'TOTAL PAY $',RP+OP
1ØØ     FORMAT(/,22X,A,F7.2)
        WRITE(*,3Ø)'ANOTHER RUN (Y/N)? '
        READ(*,11Ø)A
11Ø     FORMAT(A1)
        IF(A.EQ.'Y')GOTO 2Ø
        END
```

Again I emphasize that you should not change the *logic* of the program. You should change the *structure,* or appearance. If you have time, you may want to modify the program to include deductions like Social Security and Federal Income Tax in the table.

3. Return to the Income Tax program you wrote at the end of the last chapter. Spruce it up using the Structured Programming techniques you've learned. You should now be able to use more descriptive variable names (since you've now learned how to change the default variable types), add comment lines, and perhaps use the new logical operators ".AND." and/or ".OR." (sounds as if I'm stuttering, doesn't it?). I hesitate to ask you to use the new "E" format (which would be a bit unconventional in a financial report), but, if you did, you'd incorporate virtually every topic discussed in this chapter!

Chapter 11:
LOOPS

A. SENDING OUTPUT TO THE PRINTER

1. Transferring with the Filer

You should recall that it is possible to transfer an entire Text file to the Printer with the Filer. In case you've forgotten how, type **"T"** for Transfer, then respond to the "TO WHERE?" prompt with either "PRINTER:" or its volume number "#6:".

2. Control from the Program

It is also possible to write to the Printer under program control. To do so in Apple FORTRAN, one must first open the Printer as a file. Here is the command which will do so:

```
OPEN (6, FILE = 'PRINTER:')
```

or

```
OPEN (6, FILE = '#6:')
```

Notice the *single quotes* and the *colon*. The "6" in the OPEN is called a *device number*. The OPEN statement establishes a connection between the *program* device number "6", and the *operating system* device "PRINTER:". From now on, all we need do to send output to the Printer is use "6" instead of "*" in WRITE. Here's a sample use:

```
WRITE (6, 1ØØ) 'The number is ', NUMBER
```

You may switch back and forth between Printer and Console as often as you like in a single program. Just use "6" for all WRITEs to Printer and "*" for all WRITEs to Console. We will further discuss the OPEN statement when we study Data files.

B. LOOPS

Loops are used to repeat a section of program code a specified number of times. They may also be used in conjunction with a conditional to repeat code an indefinite number of times.

1. Form

```
DO line number  Integer variable = expression, expression, expression
```

NOTE: The variable must be an Integer variable. The expressions must also be of type Integer.

The variable serves as a counter. The first and second expressions assign the beginning and ending values for the counter; the *optional* third expression specifies the step (which is otherwise understood as 1).

The line number indicates the last line to be included in the loop. A loop that begins "DO 1ØØ. . ." can be read as "Do all lines from here *up to and including* line 100." When the final line of the loop is reached, the counter variable is incremented, and the loop is repeated. This continues until the counter becomes larger than the specified ending value. At this point, program control is transferred to the line immediately after the loop.

2. Examples

STATEMENTS	COMMENTS
`DO 1ØØ INDEX = 1, 1Ø`	understood step of one
`DO 45Ø NUMBER = 1, 3Ø`	
`DO 3ØØ INDEX = 1Ø, 1, -1`	you may count backwards
`DO 25Ø NUMBER = 2Ø, 1Ø, -2`	
`DO 1Ø5 SUBSCR = FIRST, LAST`	using variables is OK;
`DO 4ØØ COUNTR = 1, LAST, STEP`	use INTEGER statement if needed, though

3. CONTINUE

CONTINUE is a dummy statement which is not executed. It is primarily used as a consistent statement to label as the final statement of a loop. The following examples are *equivalent:*

```
      DO 1Ø COUNTR = 1, 1Ø
         SUM = SUM + COUNTR
1Ø    CONTINUE

      DO 1Ø COUNTR = 1, 1Ø
1Ø       SUM = SUM + COUNTR
```

In the above examples, "COUNTR" and "SUM" would have been declared as Integer variables.

The use of CONTINUE facilitates the readability of loops, since it pairs with DO as a visual bracket. Indenting the body of a loop also makes a program easier to read. Get into the habit of doing so!

4. Sample Program

The following sample program and the associated text should help tie together what we've learned so far about FORTRAN and its idiosyncrasies.

This program is to produce a table of Integer values along with the square and cube of each value. The program user will supply starting and ending values for the table. The program contains no comment statements, since it is discussed at length in the text.

```
      PROGRAM TABLE
      INTEGER FIRST, VALUE
      CHARACTER REPLY*1
```

Note the use of descriptive variables. Variable "LAST" is also used, but happens to be an Integer variable by default. (It would cause no error to place "LAST" in the "INTEGER" statement. Some programmers like to list *all* variables at a program's beginning, regardless of default.)

```
      WRITE (*, 5) '>>> Produces a table of squares and cubes <<<'
5     FORMAT (/, A)
1Ø    WRITE (*, 2Ø) 'First value to include in table (I3) ? '
2Ø    FORMAT (/, /, /, A, $)
      READ (*, 3Ø) FIRST
3Ø    FORMAT (I3)
```

It is crucial that you specify the FORMAT to be used when asking for input. The program assumes the program user understands what is meant by "I3". Line 20 skips three lines, prints a message, and "$tays put", awaiting the response. Note that the I3 FORMAT in line 30 assures that no number

larger than 999 will be accepted. (If you don't know why, read "I Format with READ" in Chapter 8.)

```
        WRITE (*, 35) 'Last value to include (I3) ? '
35      FORMAT (/, A, $)
        READ (*, 3Ø) LAST
```

Notice that a previously specified FORMAT was re-used.

```
        IF (FIRST .GT. LAST) THEN
           WRITE (*, 5) 'Oops, try again.'
           WRITE (*, 5) 'First value should be < last value!'
           GOTO 1Ø
        ENDIF
```

Note the use of an error trapping block.

```
        WRITE (*, 4Ø) 'Number', 'Squared', 'Cubed'
4Ø      FORMAT (/, /, /, A, 3X, A, 5X, A)
```

The table heading appears after skipping three lines.

```
        DO 6Ø VALUE = FIRST, LAST
           WRITE (*, 5Ø) VALUE, VALUE**2.Ø, VALUE**3.Ø
5Ø         FORMAT (1X, I3, 5X, F7.Ø, 3X, F1Ø.Ø)
6Ø      CONTINUE
```

The loop does the bulk of the work in the program. The loop index ranges between the two values supplied by the program user, and the WRITE statement prints the index's value, its square, and its cube.

You may wonder why the "F" FORMATs are used in line 50 if we're working with Integers. This is because the maximum Integer value is 32,767; but 200 squared is already 40,000. If we're allowing values up to 999, Integer values cannot cover their squares, much less their cubes! To achieve Real values, we must take the Integer base (VALUE) to a Real power (2.0 and 3.0); mixed operands yield Real values, you should recall.

It is important to anticipate the maximum field width the output will require. If we declare a width which is too small, all we will see is a row of asterisks.

```
        WRITE (*, 2Ø) 'Another table (Y/N) ? '
        READ (*, 7Ø) REPLY
7Ø      FORMAT (A1)
```

```
            IF (REPLY .EQ. 'Y') GOTO 1Ø
C           Otherwise
            END
```

The format used to accept the reply is "A1", which accepts only the first character of input. Replies of "Y", "YES", and "YUP" would all result in another run.

5. Review Questions

1. Do you think the Printer could be opened with *program* device number 8, or must it always be 6 (since the *operating system* device number is #6:)?

2. What would happen if a loop's counter variable was given a starting value larger than its ending value?

3. List the values each of the six example loops' (see section B.2, "Examples") counters would have *after* the loops have been executed.

4. How would one write a loop with a fractional step if all of DO's indices must be Integers?

6. Exercises

1. Suggestions for modifying the sample program:

- **a.** The format for the "CUBED" column is "F10.0", or nine digits plus the decimal. Apple FORTRAN has only six significant digits, therefore the last three digits are always zeroes. Try using an "E" FORMAT for this column instead.
- **b.** Make a similar table containing a number, its square root (VALUE**0.5), and cube root (VALUE**0.333333). This would also entail devising different output FORMATs in lines 40 and 50.
- **c.** Send the table to the Printer rather than to the Console. The user prompts should still appear on the Console.

2. Refer back to the GASUSE and SIREN programs you entered in the Introduction. You should now have no trouble understanding GASUSE. There are still a few "mystery lines" in SIREN, but you should be able to comprehend the purpose of the three loops. What values will each of the

counters process? Add a WRITE statement to the body of each loop in order to print the values of the counters. Was your analysis correct?

3. The SIREN program contains an example of *nested loops* (one or more loops contained within the body of another loop). When two loops are nested, the inner loop's index cycles all the way from its beginning to its ending value *each time* the outer loop's index changes. Make use of this behavior in writing a program to produce a multiplication table for your 4th-grade brother. (Okay, so pretend you *do* have a younger brother!) It should list all the multiplication facts from 1 to 12. The beginning and ending lines should look like this:

```
 1 X  1 =   1

 1 X  2 =   2

 1 X  3 =   3









12 X 10 = 120

12 X 11 = 132

12 X 12 = 144
```

NOTES:

1. Do not use "*" as a multiplication symbol in the table. (We programmers tend to forget that our younger brothers and sisters are still using "x ")

2. It would be wise to use a *delay loop* to slow the output to 4th-grade reading speed. A delay loop is a loop with no body. (Sounds like a shampoo commercial, doesn't it?) Its purpose is simply to make the loop index count to a number somewhere in the thousands, in order to delay the program for a few seconds. Note that FORTRAN has no command analogous to Applesoft BASIC's "SPEED".

3. Several blank lines should separate each group of twelve items.

Chapter 12:
ARRAYS

An array is a list of related values which share a common name. Individual elements in the array are distinguished by a numerical subscript, which is supplied after the array name.

A. NAMING ARRAYS

Subscripted variables in FORTRAN are written exactly as they are in BASIC:

```
array name (subscript)
```

The subscript must be a positive Integer or Integer expression (no Reals). The variable used to name the array must of course contain six or fewer characters. Here are some valid examples:

```
AVERGE(3)     TEMP(23)     BILLS(INDEX)     COINS(NUMBER * 2)
```

B. ARRAY TYPES

Arrays are either all-Integer, all-Real, or all-Character, depending on the variable used to name them.

`MAX(ITEM)`	is an all-Integer array
`AVE(INDEX)`	is an all-Real array
`NAME(ITEM)`	is an all-Character array, assuming NAME was declared a Character variable at the start of the program.
`FRAME(COUNTR)`	is an all-Integer array if FRAME was declared as an Integer variable at the start of the program. Note that COUNTR would also need to be declared as an Integer.

C. MISCELLANEOUS POINTS

1. *All arrays must be dimensioned in FORTRAN,* regardless of their size. Subscripts begin at 1 (not 0!). The keyword needed to declare an array is DIMENSION:

```
DIMENSION PRICE(9Ø)
```

PRICE is now an all-Real array with room for 90 elements.

```
DIMENSION PLAYER(3Ø), POINTS(3Ø)
```

More than one array may be specified in a single DIMENSION statement.

2. Apple FORTRAN allows arrays of up to three dimensions. The corresponding DIMENSION statement for such an array might look like this:

```
DIMENSION STUDY (2Ø, 1Ø, 5)
```

CAUTION: Three-dimensional arrays use deceivingly large amounts of memory. The above array contains 1000 elements! Fortunately, many programmers seldom (if ever) need to use arrays with three subscripts.

D. INITIALIZING ARRAYS

A quick way to initialize arrays is with the DATA statement:

```
DATA VALUE / 1, 2, 3, 4, 5, 6, 7, 8, 9, 1Ø, 1Ø, 1Ø, 1Ø /


DATA SUIT / 'Clubs   ', 'Spades  ', 'Hearts  ', 'Diamonds' /
```

NOTE: It is not necessary to supply any subscripts in "DATA"; one simply lists the array name, followed by the initial values for each element of the array.

In the second example above, 'Clubs ' would be SUIT(1), 'Spades ' would be SUIT(2), etc.

E. ORDERING OF STATEMENTS

DIMENSION must appear before any executable lines. Here is the ordering enforced in every program by our Compiler:

STATEMENT	COMMENTS
PROGRAM	always the first line of a program
IMPLICIT	second on the "ladder"
INTEGER, REAL, CHARACTER, DIMENSION	these four have equal "rank"
DATA	fourth rung
E X E L C I U N T E A S B L E	
END	always the last line of a program

NOTE: Although "DIMENSION" may be placed before "REAL", "CHARACTER", and "INTEGER", it is not logically sound to do so. You should be able to see why from this example:

```
DIMENSION NAME(3Ø), COURSE(3Ø)
CHARACTER NAME*25, COURSE*25
```

Is NAME a Character array (since NAME is also declared in a CHARACTER statement) or is it an Integer array (since NAME is an Integer variable by default, and the CHARACTER declaration comes *after* the DIMENSION)? Likewise, is COURSE a Character or Real array? To avoid any possible confusion, simply place the CHARACTER declaration *before* DIMENSION.

F. REVIEW QUESTION

In the Chapter text (section D, "Initializing Arrays"), values were assigned to arrays VALUE and SUIT with a DATA statement. List the type statements (INTEGER, REAL, and/or CHARACTER) and DIMENSION statements which are required to precede DATA for these two examples.

G. EXERCISES

Loops are very useful for changing array subscripts when reading or writing entire arrays. Loops are also used to search sequentially through the items in an array, again simply by letting the loop index vary the subscript.

Parallel arrays are groups of two or more arrays in which each item in a given array corresponds to (or parallels) the item(s) with the same subscript in the other array(s).

We will use loops and parallel arrays in our two exercises.

1. Use seven parallel arrays (one Character array and six Integer arrays) to accept a list of ten baseball players' names, as well as their numbers of at-bats, singles, doubles, triples, home runs, and RBIs (runs batted in).

After these seven items have been typed by the program user for each of the ten players, the program should ask its user which player he or she would like to see the stats for. Once a name has been chosen, search through the array of names until you find the requested one; then, using the same subscript with which you found the player's name, report that player's statistics by retrieving the parallel items from the other six arrays. (Or, if the name was not found in the ten player list, simply say so.)

Keep this program on your disk when finished, for we will expand upon it after we cover two more chapters.

2. Write a program to produce a student grade report as follows. Ask for a student name, followed by the number of courses in which he or she is enrolled (up to a limit of 10). Then, for each of the courses, accept the following items: course name (up to 25 letters), course instructor (up to 15 letters), number of credit hours for the course, and the letter grade earned (gasp!). I hope you see the need for four parallel arrays.

After this information has been typed, print the grade report *on the Printer,* including the student's grade point average, where GPA = (total quality points/total credit hours). Quality points are determined by multiplying the credit hours for a course times the 4-point scale equivalent of the letter grade (an "A" is 4 points, "B" is 3, etc.). A suggested format for the report follows:

STUDENT : Joe Fortran

COURSE NAME	INSTRUCTOR	HRS	GRADE	Q PTS
Learning Apple Fortran	Geenen	3	A	12
Astrophysics	Einstein	4	C	8
Calculus	Newton	4	A	16
English Composition	Shakespeare	3	B	9
Greek Philosophy	Plato	3	B	9
	TOTALS - Hours :	17	Quality Pts :	54

GPA : 3.176

Joe's school certainly has an illustrious faculty, doesn't it? (Cough, cough!) Note that it is not necessary to save the quality points for each course in an array, for they can be calculated as the report is being printed.

Chapter 13:
FUNCTIONS

A function's purpose is to receive one or more values as arguments, manipulate these values (usually by plugging them into some kind of equation), and then return a single value (the answer to the equation). Now that you know the function of a function (groan. . .), let's discuss the three forms in which they occur in FORTRAN.

A. INTRINSIC FUNCTIONS

These are pre-defined functions which can be invoked at any point in a program. Recall that every program you execute has two units of the System Library linked into it. One of the units contains the Intrinsic functions, so any program may reference these functions by name.

As is true in BASIC, FORTRAN functions consist of a short name followed by a list of arguments in parentheses. The function name actually doubles as a variable, since it serves to store the answer to the function equation or table. The function name "variable" then returns that answer to the program line which called the function. The individual arguments may be as simple as a single number or variable, or a complex expression. Here is a list of the majority of functions available in Apple FORTRAN:

FUNCTION	KEYWORD	ARGUMENT(S)	RETURNS
conv. to int.	INT	real	integer
conv. to real	REAL	integer	real
ASCII to char.	CHAR	integer	character
char. to ASCII	ICHAR	character	integer
round-off	ANINT	real	real
round-off	NINT	real	integer
absolute val.	ABS	real	real
absolute val.	IABS	integer	integer
mod division	MOD	2 integers	integer
mod division	AMOD	2 reals	real
choose largest	MAXØ	integer list	integer
choose largest	AMAX1	real list	real
choose smallest	MINØ	integer list	integer
choose smallest	AMIN1	real list	real

FUNCTION	KEYWORD	ARGUMENT(S)	RETURNS
square root	SQRT	real	real
exponential	EXP	real	real
natural log	ALOG	real	real
common log	ALOG1Ø	real	real
sine	SIN	real (radians)	real
cosine	COS	real (radians)	real
tangent	TAN	real (radians)	real
end of file	EOF	integer	logical (T or F)

See the *Apple FORTRAN Language Reference Manual* if you're really interested in inverse cosines, hyperbolic tangents, and the like. For most programmers, the above list should more than suffice. Note that in general, if the function name begins with the letters I–N, it will return an Integer value.

NOTE: Please read the above table carefully, since all functions are particular as to what type of numbers or variables (Real or Integer) they act upon, and what type of values (Real or Integer) they return. There is no automatic "REAL" or "INT" called in for you when types of arguments do not match! The same applies to the other kinds of functions discussed in this lesson.

A function may be used in an assignment statement, a conditional, or a WRITE statement. Here are a few examples:

```
ANSWER = SQRT (45.Ø * X)
IF (ABS (TOTAL) .LE. 6.Ø) GOTO 4ØØ
WRITE (*, 1ØØ) TAN (ANGLE)
```

"CHAR" is a very useful function which converts a decimal (base 1Ø) ASCII code to its corresponding character. It can be used to get non-printing *control* characters such as the bell (CTL-G, or ASCII 7), form feed (CTL-L, or ASCII 12, which clears the screen, or will advance the Printer to the next page), inverse printout (CTL-O, or ASCII 15, on an Apple IIe or IIc), and normal printout (CTL-N, or ASCII 14). Here is a sample program segment using CHAR:

```
      PROGRAM SAMPLE
      CHARACTER BELL*1, HOME*1, INVRSE*1, NORMAL*1

C     Assign using "CHAR"
      BELL = CHAR (7)
      HOME = CHAR (12)
      INVRSE = CHAR (15)
      NORMAL = CHAR (14)
```

```
C       Clear screen
        WRITE (*, 1ØØ) HOME
1ØØ     FORMAT (A, $)

        :
C       Ring bell
        WRITE (*, 1ØØ) BELL
        :
        :
C       Set inverse print out
        WRITE (*, 1ØØ) INVRSE

        :
C       Return to normal print out
        WRITE (*, 1ØØ) NORMAL
```

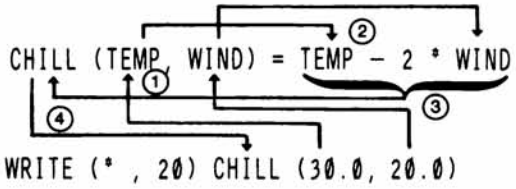
B. STATEMENT FUNCTIONS

A programmer is free to define her/his own functions. If the function can be expressed in a *single* instruction (no more than one line), then it may be defined in the Main Program *before any executable lines.* (In a moment we will recap the order for all of the so-called Specification statements.) The function must be given a name, a list of arguments (there is no limit to the number of arguments you may list), and then a definition in the following form:

```
function name (variable(s)) = expression
```

Here are some examples and explanations:

```
CHILL (TEMP, WIND) = TEMP - 2 * WIND
```



① The arguments are passed to the Statement Function. In this example, variable TEMP receives a value of 30.0, and WIND receives a value of 20.0.

② The newly received values are substituted into the function definition (equation). In this case, the expression becomes 30.0 − 2 * 20.0.

③ The equation is evaluated and its value is stored in the function name. Therefore, CHILL has a value of −10.0.

④ The function name, acting as a variable, returns its value to the program line from which it was called.

Figure 13.1 How a Statement Function Operates

This function computes a simplified version of what we Wisconsinites call the "windchill factor." (The thermometer reading alone somehow doesn't do our winter climate justice.) It has the name CHILL, and requires two *Real* arguments (TEMP and WIND). After plugging these arguments into the function definition (TEMP − 2 * WIND), it will return a *Real* value, since CHILL begins with a "C".

```
NEWSUM (I, J, K) = I + J + K
```

NEWSUM requires three Integer arguments, and will return an Integer value since it begins with an "N".

At last we know all the SPECIFICATION STATEMENTS available in FORTRAN. These statements define variable types, arrays, and functions. They must come before any executable lines, and may not have a line number (in other words, they cannot be the target of a GOTO). Here they are, along with their corresponding "rank":

1. PROGRAM
2. IMPLICIT
3. CHARACTER, REAL, INTEGER, DIMENSION (any order)
4. DATA
5. statement function definitions

The reason Statement Functions must come last is because any change of default variable types may affect the type of value the function will return. In addition, a function may not have the same name as an array.

C. SUBPROGRAM FUNCTIONS

Any function that requires more than one line of code must be defined in a *Subprogram Function.* This series of lines is placed *after* the "END" statement of the Main Program. It will then be compiled *separately* from the Main Program.

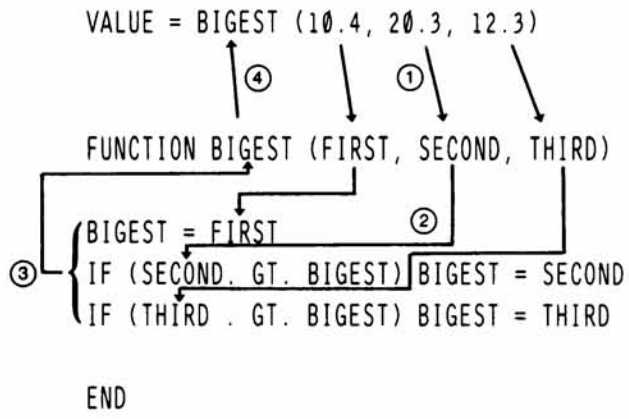
Subprogram Functions usually do not perform any input or output (although it is perfectly legal for them to do so). They merely serve to carry out calculations or manipulate items in memory, then return a single value to the Main Program.

As with Statement Functions, Subprogram Functions need a name, followed by one or more arguments in parentheses. A Subprogram Function must begin with the keyword FUNCTION and end with the word END, as follows:

```
FUNCTION name (variable(s))
:
function code
:
END
```

A Subprogram Function may even have its own set of specification statements, so it literally is a mini-program! Here is an example:

```
FUNCTION BIGEST (FIRST, SECOND, THIRD)

BIGEST = FIRST
IF (SECOND .GT. BIGEST) BIGEST = SECOND
IF (THIRD .GT. BIGEST) BIGEST = THIRD

END
```



① The arguments are passed to the Subprogram Function.
② These values are substituted into the function code.
③ The function code is evaluated; this often entails several steps. *Note that the function name is used as a variable.* It must be assigned a value, for it sends its value back to the Main Program.
④ The function name, acting as a variable, returns its value to the Main Program line from which it was called.

Figure 13.2 How a Subprogram Function Operates

The above function is named BIGEST, requires three Real numbers as arguments, and will return the largest of these three Real numbers. (Aside: Alert readers may have noticed that this function is already available as "AMAX1".)

NOTE: There are two channels of information exchange between the Main Program and the Subprogram Function:

1. *MAIN PROGRAM TO FUNCTION*

This is done via the arguments listed in parentheses. These values are sent to the function by the Main Program. They may be constants, variables, or expressions. They MUST match in type with those values expected by the function.

2. *FUNCTION TO MAIN PROGRAM*

This is done via the function name. The name of the function doubles as a variable, and as such it sends a value back to the Main Program. Therefore, EVERY FUNCTION MUST USE ITS NAME AS A VARIABLE!

How does one summon the above function from the Main Program? Exactly the same way one summons any other function. Here are a few ways to do so:

```
VALUE = BIGEST (1Ø.4, 2Ø.3, 12.3)

WRITE (*, 1ØØ) BIGEST (ONE, TWO, THREE)

IF (BIGEST (1Ø*X, 2Ø*Y, 3Ø*Z) .LT. Ø.Ø) GOTO 4ØØ
```

D. CHANGING SUBPROGRAM FUNCTION TYPES

You may change the default type (Real or Integer) of your Subprogram Function by using the following form:

```
type FUNCTION name (variable(s))
```

For example:

```
INTEGER FUNCTION DOLE (COINS, MONEY)
```

DOLE would normally return a Real value.

```
REAL FUNCTION JUMBO (NUMBER)
```

JUMBO now returns a Real value.

NOTE: Functions may not be of type "CHARACTER".

E. REVIEW QUESTIONS

1. What are specification statements?

2. A person receives several error messages for a program that begins and ends as follows:

```
1Ø      PROGRAM ERROR
        CHARACTER REPLY*1
        IMPLICIT INTEGER (A - H)
        DIMENSION MISSES (2Ø)



        :
        IF (REPLY .EQ. 'Y') GOTO 1Ø
C       Otherwise
        END
```

Can you find and correct the errors?

3. There were three functions defined in the text of the chapter. Using their definitions, give the answers that will be printed by the following function calls (assume the FORMATs referenced are appropriate):

a. `WRITE (*, 5Ø) CHILL (25.Ø, 7.5)`

b. `WRITE (*, 5Ø) CHILL (7.Ø, 15.Ø)`

c. `WRITE (*, 1ØØ) NEWSUM (1Ø, 23, 9)`

d. `WRITE (*, 1ØØ) NEWSUM (6.5, 1.5, 4.Ø)`

e. `WRITE (*, 15Ø) BIGEST (13.Ø, -25.2, 19.7)`

f. `WRITE (*, 15Ø) BIGEST (6.1, 12.8, Ø.Ø)`

F. EXERCISES

1. Write a short program that will convert a given number of inches to centimeters. Use a Statement Function to do the actual conversion. There are 2.54 centimeters in an inch.

2. It is possible for one function to call another function. You may have noticed that the arguments to the trig functions (SIN, COS, and TAN) must be expressed in radians. If you wrote a Statement Function to convert

an angle measured in degrees to radians (multiplying degrees by Ø.ØØ1745 yields radians), you could then call the trig functions with the *result* of this conversion function:

```
COS ( RADIAN (ANGLE) )
```

where RADIAN is the name of your Statement Function, and ANGLE is the angle measured in degrees.

Use this technique in a program which will provide its user with a trig function menu, then ask for the angle with which the chosen function should be evaluated.

3. Write a program to compute Miles per Gallon given the car's odometer reading at the last fill-up, the current odometer reading (subtracting these two odometer readings will give you the miles travelled), the amount of gas purchased (in dollars), and the price per gallon of the gas (dividing the amount by the price will give you the gallons used). Use a Subprogram Function to perform the calculations and return the mileage. You may want to use the CHAR function to help you clear the screen, print in inverse, etc.

Chapter 14:
SUBROUTINES

A. FORM AND EXAMPLES

A subroutine, like a Subprogram Function, is an *independently compiled* set of instructions placed after the "END" statement of the Main Program. The subroutine form is:

```
SUBROUTINE name (variable(s))

subroutine code

END
```

The subroutine is called by this statement:

```
CALL name (variable(s))
```

In both cases, "name" is any valid FORTRAN variable.

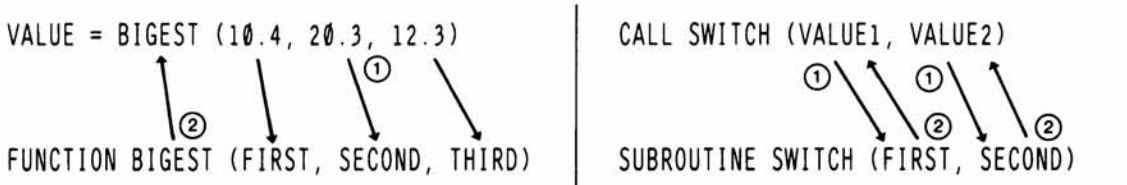
NOTE: The variables in the CALL and SUBROUTINE argument lists must match in number and in type (Integer, Real, or Character). As is true with functions, there is no automatic "INT" or "REAL" called in when argument types don't match. When you pass Character arguments from a Main Program to a subroutine, the variables in both units must be EXACTLY THE SAME LENGTH.

Perhaps an easier way to visualize the format is by an actual example:

```
PROGRAM SCORES

:
CALL SWITCH (VALUE1, VALUE2)

:
END
```

```
SUBROUTINE SWITCH (FIRST, SECOND)

TEMP = FIRST
FIRST = SECOND
SECOND = TEMP

END
```

The above subroutine will exchange the values of two variables. The Main Program sends two Real values to the subroutine (VALUE1 and VALUE2). These two values are received by the subroutine as FIRST and SECOND. *It is important to note that although the Main Program and the subroutine use different variable names, they are both referring to the same two values.* In other words, variable VALUE1 (from the Main Program) and variable FIRST (from the subroutine) correspond to the *same* memory location. Therefore, when SWITCH changes the values of FIRST and SECOND, it is changing the values of VALUE1 and VALUE2 also.



A function ① receives one or more arguments, then ② returns a *single answer* via the function name.

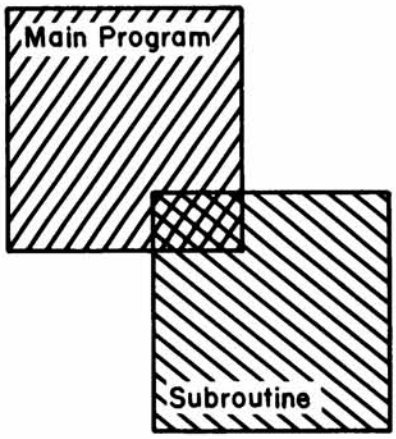
A subroutine ① receives one or more arguments, then ② returns *the same number of values it was sent.* In the above example, 2 arguments were sent to the subroutine, and 2 arguments were returned by the subroutine.

Figure 14.1 The Difference between a Function and a Subroutine

NOTE: THE VARIABLES USED IN ANY SEPARATELY COMPILED UNIT (A MAIN PROGRAM, A SUBPROGRAM FUNCTION, OR A SUBROUTINE) ARE SAID TO BE "LOCAL" TO THAT UNIT. In other words, the variables in a Main Program cannot be accessed by a subroutine (nor vice versa). In fact, when two units use the same name for a variable, they are actually referencing two independent memory locations. Another peculiarity: LINE NUMBERS ARE LOCAL TO THE COMPILATION UNIT. In other words, a Main Program and a subprogram may share identical line numbers with no possible risk of confusion.

There is an exception to the above rule. THOSE VARIABLES WHICH ARE LISTED AS ARGUMENTS OF A FUNCTION OR SUBROUTINE

ARE SAID TO BE "GLOBAL." Two units may have access to the same memory location(s) via this method. As we have just seen, the commonly referenced memory locations may even be called by different names in each of the two units.



Local variables can be thought of as occurring in the non-overlapping areas. Some variables are known only to the Main Program; others are known only to the subroutine.

Global variables can be thought of as occurring in the overlapping area. They are known to both Main Program and subroutine, even though they may happen to have different variable names in each unit.

Figure 14.2 Visualizing Global Variables

Upon completion of the subroutine, the values of the arguments (in the above case, FIRST and SECOND) are returned to the Main Program (received in this case as VALUE1 and VALUE2). In addition, program control is transferred to the line following the CALL statement.

It is important that the programmer give the subroutine arguments some value in the subroutine. Otherwise the subroutine will be useless. *REMEMBER: The only variables that both the Main Program and the subroutine have mutual access to are those listed as arguments. All other variables are LOCAL.*

B. PASSING ARRAYS AS ARGUMENTS

What if one needed to send 100 values to a subroutine? Would he or she need to list 100 arguments? Fortunately, no!

One may pass an entire array as an argument simply by listing the array name. The array must be DIMENSIONed in both the Main Program and the subroutine.

Here is a sample use of an array being passed as an argument. The Main Program sends the subroutine an unsorted 35-element array of test scores, along with the number of students (array elements) that actually need to be sorted. The subroutine sorts the specified number of elements in descending order, then returns the sorted array to the main program:

```
      PROGRAM SCORES
      INTEGER TEST
      DIMENSION TEST (35)

      :
      CALL SORT (TEST, NUMBER)

      :
      END


      SUBROUTINE SORT (EXAM, SIZE)
      INTEGER EXAM, SIZE, FIRST, SECOND, TEMP
      DIMENSION EXAM(35)

      DO 2Ø FIRST = 1, SIZE-1

         DO 1Ø SECOND = FIRST+1, SIZE

            IF (EXAM (FIRST) .LT. EXAM (SECOND)) THEN
C              Switch their values
               TEMP = EXAM (FIRST)
               EXAM (FIRST) = EXAM (SECOND)
               EXAM (SECOND) = TEMP
            ENDIF

1Ø       CONTINUE

2Ø    CONTINUE

      END
```

Why did the subroutine contain an INTEGER statement? Because without it, "EXAM" and "SIZE" would be Real variables in the subroutine's eyes. Why a DIMENSION statement? Because without it, "EXAM" would be a simple Real variable, not an array. Remember that the subroutine is compiled *independently* of the main program. Any Main Program declarations are therefore *unknown* to the subroutine!

Note how the use of blank lines and indentation helped to separate and highlight the subunits of the subroutine (the nested loops, the switching routine, etc.).

What would happen if a programmer wished to use the same subroutine with arrays of *different* sizes? Or better yet, what if the size of the array would not be known in advance? Apple FORTRAN has a rather unusual

feature to handle this dilemma. An asterisk (*) is used to indicate an *assumed-sized array.* Check out this rewritten example:

```
SUBROUTINE SORT (EXAM, SIZE)
INTEGER EXAM, SIZE, FIRST, SECOND, TEMP
DIMENSION EXAM(*)
```

Now when an array is sent to SORT, the size of the array in the Main Program will be used to set the dimension in the subroutine. Even more surprising, Apple FORTRAN will allow an assumed-size array to be re-dimensioned during the course of a single run.

In other words, you could call SORT with a 20-element array, then later with a 30-element array, without receiving an error message. This is going to become very valuable when we begin discussing the creation of a personal library of subroutines and functions (which are then saved for use in future programs) in the next chapter. A single subroutine may handle any number of elements from 2 to 1000 or more by using this special feature *(as long as the type of the array matches that of the Main Program).*

C. SENDING CONSTANTS AND EXPRESSIONS TO A SUBROUTINE

Now we will discuss a bit more about the arguments that one passes back and forth between main- and sub-programs. Is it possible to send a *constant* or *expression* rather than a *variable* to the subroutine? The answer is "Yes, BUT. . . ."

Look at this example:

```
PROGRAM MUSIC
DIMENSION NOTES(25Ø)
:
:
CALL JAZZ (NOTES, 4)

:
END

SUBROUTINE JAZZ (NOTES, BEATS)
INTEGER BEATS
DIMENSION NOTES(*)

:
END
```

Recall that the communication between program units occurs through the arguments. The problem with using a constant in the CALL statement is that if the corresponding variable in the subroutine ("BEATS" in the above example) should change, the Main Program would never "know"; there is no *variable* (only the constant "4" in the example) to receive the value returned by the subroutine. This would *not* cause an error, however, and sometimes this approach is necessary.

If you do send a constant or expression to a subroutine, just be sure it matches the type of value expected by the subroutine. If it is a Character constant, it must be EXACTLY the same length as the subroutine variable which receives it. (If it is too short, fill it out with spaces. If it is too long, declare the subroutine variable with a longer length.)

D. FUNCTIONS VS. SUBROUTINES

Beginners are often unable to see the difference between FORTRAN functions and subroutines. We'll clear up any confusion now:

1. COMMUNICATION FROM MAIN PROGRAM TO SUBPROGRAM

In both cases (functions and subroutines), access is gained to Main Program values by the list of arguments supplied with the function or subroutine. These arguments are *global;* other values, including line numbers, are *local* (not mutually available).

2. COMMUNICATION FROM SUBPROGRAM TO MAIN PROGRAM

a. A function usually returns *one and only one value* to the Main Program. This value is sent back via the function name, which acts as a variable.

b. A subroutine may return *any number of values* (from 0 to 100 or more). These values are sent via the list of variable arguments.

3. ABILITY TO MODIFY MAIN PROGRAM VALUES

Both functions and subroutines can not only access Main Program values, *they can change them.* When the values of their arguments change, so do the corresponding arguments in the Main Program!

4. ABILITY TO CALL OTHER SUBROUTINES AND/OR FUNCTIONS

Both functions and subroutines may reference other functions and subroutines, but recursive calls (a subroutine or function calling itself) are *not* allowed.

5. WHAT GOOD ARE THEY?

Both functions and subroutines alleviate the need to re-type identical code in more than one place in a program. Yet, even if there is no repeated code, subprograms can help make programming easier. They allow one to break large programs into more manageable subunits. Then the programmer may write one subprogram to handle each!

Doing so offers a number of advantages:

a. It helps focus a programmer's attention on a given task. You're worrying about only one thing at a time.

b. It makes large programs easier from a psychological standpoint. (You'd probably rather write eight short "programs" than one huge program.)

c. It allows several persons to tackle a program too large for a single person to finish in a reasonable amount of time. (Each person writes a single subroutine or function.)

d. It allows a programmer to place useful subprograms into a library, so they may be used again without the need to be rewritten.

Business programmers are often assigned one subprogram each. When the individuals have the components completed, they are combined into a single unit via the Main (Calling) Program, which may look as simple as the following:

```
      PROGRAM BILLNG

C     Get list of monthly transactions
      CALL TRNSCN (arguments)

C     Get list of customer accounts
      CALL ACCNTS (arguments)

C     Update customer accounts
      CALL UPDATE (arguments)

C     Print outstanding bills
      CALL BILL (arguments)

      END
```

E. REVIEW QUESTIONS

1. List the differences between a function and a subroutine.

2. List the similarities between a function and a subroutine.

3. Define the words GLOBAL and LOCAL. How do you determine whether a variable is global or local?

4. Why do you think line numbers are local to the compilation unit?

5. Why do you think global variables may have different names in different program units, yet still refer to the same memory location?

6. Do you think it is possible that a subroutine would need no arguments?

7. Of what use is the "*" in a DIMENSION statement?

F. EXERCISES

1. The all-time classic programming exercise involving subroutines is the change-maker program. After asking for a purchase price and amount of cash received to pay for the purchase, the program proceeds to report the amount of change to return. It then provides a coin-by-coin breakdown of the change, similar to the following:

```
Your change is $14.31

  Ø Twenty(ies)
  1 Ten(s)
  Ø Five(s)
  4 One(s)
  1 Quarter(s)
  Ø Dime(s)
  1 Nickel(s)
  1 Penny(ies)
```

Write such a program (using your vast knowledge of programming with structure and style, of course!). Hints: Where in the program is there code which is repeated? Once found, that code may be placed in a subroutine and called repeatedly. What arguments need to be passed from the Main Program to the subroutine and vice versa?

2. Recall that you had written a program using seven parallel arrays at the end of Chapter 12. The arrays each contained ten elements, and were used to store baseball players' names, at-bats, singles, doubles, triples, home runs, and RBIs. We will now modify that program.

The input section of the program need not be changed. This time, however, after accepting the list of names and stats, you will ask the program user which of the six statistics he or she would like to sort upon. If RBIs are chosen, for instance, you will sort players based upon their RBI totals (highest to lowest). You will then print each of the arrays and, after a delay, return to the selection menu.

The problem that arises when you are sorting *one* parallel array is that the corresponding elements in the other parallel arrays must be switched, too (*or else the arrays would no longer be parallel*). Whenever you exchange values in the selected (or *key*) statistic array, simply switch the corresponding values in the other arrays. You should recall that a switch of two values requires three assignment statements. To switch seven groups of two values would require twenty-one statements *unless you were wise enough to use a subroutine.* You will need *two* switching subroutines in this instance, since one of the arrays contains Character values. I will be kind enough to supply you with their code:

```
      SUBROUTINE CSWAP (FIRST, SECOND)
C     Swaps two 25 byte Character values

      CHARACTER FIRST*25, SECOND*25, TEMP*25

      TEMP = FIRST
      FIRST = SECOND
      SECOND = TEMP

      END


      SUBROUTINE ISWAP (FIRST, SECOND)
C     Swaps two Integer values

      INTEGER FIRST, SECOND, TEMP

      TEMP = FIRST
      FIRST = SECOND
      SECOND = TEMP

      END
```

You will need a third subroutine to sort out a ten-element Integer array. This will in turn call the other two subroutines when it comes time to swap values. I'm still in a generous mood, so I hereby unveil the code:

```
      SUBROUTINE SORT
     *    (KEY, FIRST, SECOND, THIRD, FOURTH, FIFTH, NAME)
```

```
C       Descending sort of KEY array, with parallel Integer arrays FIRST,
C       SECOND, THIRD, FOURTH, FIFTH, and parallel Character array NAME.

C       Specification statements
        IMPLICIT INTEGER (A - Z)
        CHARACTER NAME*25
        DIMENSION KEY(1Ø), FIRST(1Ø), SECOND(1Ø), THIRD(1Ø),
     *            FOURTH(1Ø), FIFTH(1Ø), NAME(1Ø)

C       Sorting routine

        DO 2Ø ITEM1 = 1, 9

           DO 1Ø ITEM2 = ITEM1 + 1, 1Ø

              IF (KEY (ITEM1) .LT. KEY (ITEM2)) THEN
C                Switch items in all parallel arrays
                 CALL ISWAP (KEY (ITEM1), KEY (ITEM2))
                 CALL ISWAP (FIRST (ITEM1), FIRST (ITEM2))
                 CALL ISWAP (SECOND (ITEM1), SECOND (ITEM2))
                 CALL ISWAP (THIRD (ITEM1), THIRD (ITEM2))
                 CALL ISWAP (FOURTH (ITEM1), FOURTH (ITEM2))
                 CALL ISWAP (FIFTH (ITEM1), FIFTH (ITEM2))
                 CALL CSWAP (NAME (ITEM1), NAME (ITEM2))
              ENDIF

1Ø            CONTINUE

2Ø      CONTINUE

        END
```

Note that Subroutine SORT has seven arguments (the seven parallel arrays). Since these arguments are global, the changes that take place in SORT will be reflected in the Main Program. Also note the seven CALL statements that SORT makes to either ISWAP or CSWAP.

You now have a very solid start on the program. Those looking for a challenge may want an option to sort the names alphabetically, or may wish to add an eighth array containing the batting averages. (Why won't it be necessary to have this array's values entered by the program user?) Good luck!

Chapter 15:
CREATING A PERSONAL LIBRARY

Apple FORTRAN allows one to incorporate previously created functions or subroutines into another program at three different stages of program development: Editor, Compiler, and Linker.

A. COPY

The Editor contains a COPY option. Its prompt looks like this:

```
COPY: B(UFFER  F(ROM FILE  <ESC>
```

1. COPY BUFFER

The Buffer option doesn't apply directly to this lesson, but it can be extremely useful, so we'll discuss it briefly.

A *buffer* is best described as a temporary slice of memory that serves as the Editor's trash can. The contents of the buffer are subject to disposal at any time, but until disposal these contents are retrievable.

When are the contents of the buffer disposed of? Whenever there is a new Insertion or Deletion in the file. What replaces the old contents? The character(s) that were newly Inserted or Deleted.

In other words, the buffer contains all characters from one CTL-C to the next. (Recall that CTL-C makes an Insertion or Deletion permanent. We can now also add that CTL-C changes the buffer by disposing of the previous contents and replacing them with the new.)

How large is the buffer? Any size from one character to an entire file, depending on the size of the Insertion or Deletion. (Technical aside: the Editor has a limit of thirty-four blocks of room for your program plus the buffer; it will warn you of a "Buffer Overflow" in the unlikely event you ever place too many characters in it.)

We can use the buffer to our advantage when it comes time to shift a program line or two around. *Move the cursor to the line you want to move, D(elete it (do you remember how to delete an entire line all at once?), and press CTL-C (remember that it is saved in the buffer). Now move the cursor to the new position you had in mind for the line, and choose the C(OPY*

B(UFFER option. The buffer is now inserted at the current cursor position. You've just moved your line!

In the same way, you can reposition entire subunits like loops or If. . .Then. . .Elses in a program without having to re-type. Just delete them, move the cursor to a new position, and copy from the buffer.

This kind of buffer use is the heart of word-processing. Imagine being able to reposition words, sentences, paragraphs, or even whole chapters with no annoying scratch-outs! Newspapers are written almost entirely in this manner, and a growing number of books are, too (including the one you're reading).

2. COPY FROM FILE

The F(ROM FILE option of C(OPY is what we're really interested in here. Suppose you have a function or subroutine that would very likely be usable in more than one program (for example, one that alphabetizes names). If you have that subroutine saved on your disk in its *Text* version, this option will allow you to read in that entire Text file at the current cursor position.

Let's say that you've just finished typing a new program and the cursor is immediately below the "END" statement. You'd like to read in your subroutine file at this point in order to make the program complete. Here is the prompt along with your (**emphasized**) reply:

```
FROM WHAT FILE [MARKER,MARKER] ? FORTRAN:ALPHA
```

The Marker sub-option is not usable in this instance. We'll discuss it in the word-processing lesson (Chapter 20).

File FORTRAN:ALPHA.TEXT would be the diskfile name of your subroutine. Before answering this prompt, you would have had to make sure disk FORTRAN was in Drive 1 while FORT2 was in Drive 2 (why?). After copying the text of the subroutine, the Editor will ask you to press the RETURN key to get the prompt line back.

The only inherent hassle in using a program segment such as an alphabetizing subroutine in several programs is the various array sizes such a subunit must be able to handle. *The "*" dimension declarator is invaluable in this regard.*

By keeping Text versions of useful functions and subroutines, you will always have them available for use at any time without the need for retyping. In this manner, you create your own file library!

B. "$INCLUDE" COMPILER DIRECTIVE

At the Compiler level, much the same thing is possible as was just discussed for the Editor. You may call in a previously created Text file *while compiling* with the use of a Compiler Directive.

A Compiler Directive in Apple FORTRAN is signalled by a "$" in Column one of a program line.

The Compiler Directive we are discussing here looks like this:

```
$       INCLUDE filename
```

Technically, the "INCLUDE" may immediately follow the "$" (just as a comment may immediately follow the "C" in Column one), but I feel it is more consistent to place it in the same column as the rest of the program lines. For the sake of discussion, however, it will be referred to as "$INCLUDE".

NOTE: The file name by "$INCLUDE" must first be transferred to FORT1 before compiling commences.

Let's set up an example of its use. We are compiling the workfile, which contains a "$INCLUDE FORT1:FUNCTIONS.TEXT" in line 5. The first four lines of SYSTEM.WRK.TEXT would be compiled as usual, after which file FORT1:FUNCTIONS.TEXT would be read in and compiled *in its entirety* before the remainder of the workfile (lines 6 and on) would be translated.

The $INCLUDE Directive has a built-in limitation: *You may place it anywhere in the program except after "END".* This precludes compiling a program with a $INCLUDE at its conclusion (thereby incorporating a subroutine or Subprogram Function). In other words, you can read in and compile a separate Text file only in the *middle* of the compilation.

Because it is so similar to the Editor's C(opy option, and because it cannot be used at the end of a program, $INCLUDE will be of limited use.

C. "$USES" COMPILER DIRECTIVE

The third method of adding previously created material to a FORTRAN program is the most efficient. A previously compiled program subunit (i.e., a Code file) may be added to another Code file by the Linker.

This approach requires two steps. First, a Compiler Directive must be issued by the Main Program to inform the Compiler that a missing unit (either a function or a subroutine) will be spliced in *after* compilation by the Linker. As with all Compiler Directives, the "$" must appear in Column one. Its form is:

```
$       USES Uunitname IN filename
```

where "unitname" is the name of the function or subroutine which is to be linked and "filename" is its operating system file name.

This directive must be the first line of a file (even before "PROGRAM").

Normally the Compiler would not let you call a function or subroutine unless it was included at the end of the Main Program. The $USES Directive overrides this potential problem.

Here are a few samples:

```
$       USES SORT
```

This informs the Compiler that the Linker will later splice in unit SORT from FORT1:SYSTEM.LIBRARY. (*Notice that if the "IN" option is not used, the unit is assumed to be in the System Library. In addition, the "U" need not appear before the unit name.*) We will discuss some useful System Library units in Chapters 18 and 19.

```
$       USES UALPHA IN FORT1:ALPHA.CODE
```

This example informs the Compiler that subroutine ALPHA, contained in file FORT1:ALPHA.CODE, will be joined to the program by the Linker.

NOTES:

1. *THE UNIT TO BE LINKED MUST BE COMPILED. Recall that this is possible since functions and subroutines are compiled separately from Main Programs. When you have a unit ready, compile it and save it immediately. (Don't link it!)*

2. *THE "$USES" DIRECTIVE REQUIRES YOU TO MANUALLY COMPILE AND LINK YOUR MAIN PROGRAM. Recall that the "RUN" option answers all Linker prompts for you; this disallows typing an extra library file. To initiate compiling a program containing "$USES", type "C" to COMPILE (not "R" to RUN), then "L" to LINK.*

3. *THE FILE THAT YOU SPECIFY IN "$USES" MUST FIRST BE TRANSFERRED TO FORT1 BEFORE YOU MAY COMPILE. If it isn't, you will receive the error message "CAN'T OPEN FILE".*

Once you've gotten the Main Program past the Compiler (with the help of $USES), you're ready for the second step: linking the previously compiled library unit. (Remember, $USES "promises" the Compiler that this will happen.) This is easily done. Let's continue with the second "$USES" example listed above, assuming you're using a workfile. Here are the Linker prompts and your replies:

PROMPTS AND REPLIES	COMMENTS
HOST FILE? <RETURN>	opens FORT1:SYSTEM.WRK.CODE
LIB FILE? *	short for FORT1:SYSTEM.LIBRARY
LIB FILE? FORT1:ALPHA	the SECOND library file (.CODE assumed)
LIB FILE? <RETURN>	no more library files
MAP FILE? <RETURN>	no map file
OUTPUT FILE? <RETURN>	save linked version as SYSTEM.WRK.CODE

NOTE: Again, the above prompts will appear only if you manually link, which you must do to add a second library file.

It is also possible to link more than one previously created Code unit per program. For each unit you want to add, use a *separate* "$USES" Compiler Directive, and answer the "LIB FILE?" Linker prompt as many times as necessary.

D. "$XREF" COMPILER DIRECTIVE

The last Compiler Directive we will discuss has nothing to do with creating a library of useful files, but as long as we are on the topic of Directives, it will be presented. It looks like this:

```
$       XREF
```

As usual, the "$" must appear in Column one, and this Directive *must be placed before the "PROGRAM" statement.* Recall that "$USES" shares this requirement, so if one uses both "$XREF" and "$USES", *both* must appear (any order) before the "PROGRAM" statement.

This Directive causes the Compiler to provide a cross-referenced listing of variables used in the program, indicating all the lines in which each variable occurs. This is a useful debugging tool for large programs. You will find that "$XREF" has no effect unless you choose to create a Compiler Listing File. When the "Listing File ? " prompt appears, respond with either "#1:" or "#6:" to view the cross-referenced listing.

E. SUMMARY

We close this lesson with a summary of the positions where the three Compiler Directives may occur:

COMPILER DIRECTIVE		PLACEMENT
$	XREF	only before PROGRAM
$	USES USORT IN FORT1:SORT.CODE	only before PROGRAM
	PROGRAM DEMO	
	:	
$	INCLUDE FORT1:SPECS.TEXT	anywhere between PROGRAM and END
	:	
	END	

F. REVIEW QUESTIONS

1. What is a buffer, and what is it used for?

2. Briefly describe three methods available in Apple FORTRAN for incorporating previously created material in a new program.

3. When one chooses the C(OPY F(ROM FILE option, why must the disk containing the file to be copied be in Drive 1, while FORT2 must be in Drive 2?

4. When one uses $INCLUDE, why must the file to be included be present on FORT1 in Drive 1, while FORT2 is in Drive 2?

5. When one uses $USES (no pun intended), why must the file to be used be present on FORT1 in Drive 1, while FORT2 is in Drive 2?

6. What type of program text might be incorporated with the "$INCLUDE" Compiler Directive?

G. EXERCISES

1. Load one of your previously written Text files into the Editor. Experiment with the C(OPY B(UFFER option until you feel comfortable moving text around. Try copying the same buffer repeatedly. Does copying the buffer "exhaust" it, or could it be copied an indefinite number of times?

2. Use the C(OPY F(ROM FILE option until you become familiar with its use.

3. In order to gain some experience with the process of linking Code library files to a Main Program, we'll start on a very small scale. Write three short subroutines, one to print your name, another to print your address, and a third to print your phone number. Edit and compile only one subroutine at a time. After you have compiled and saved all three files, enter a Main Program to call the three subroutines. (Don't forget the $USES directives!) Compile this Main Program, then use the Linker to join it with your three Code library files. Execute the linked program.

Note: In all honesty, this is an incredibly *poor* application of Code library creation. A task of such limited scale does not lend itself to the library approach. However, the point of the exercise is not to try to emulate a corporate programming team, but rather to "learn the ropes" and become comfortable with the steps required for the process.

4. Recall the FORMAT program given in Chapter 8. It contains five subroutines, all of which could have been developed separately and placed in a library, awaiting completion of the entire project. Let's simulate doing so now.

a. Enter the Editor and write SUBROUTINE IFORMT. When finished (*don't forget—only one RETURN after END*), choose to **Q**(UIT THE EDITOR and then **W**(RITE TO A FILE: *do not create a workfile.* When the Editor asks you for the output file, answer with "FORT1:IFORMAT". (Note: You should already know that operating system filenames are *not* restricted to just six characters, even though subroutine names are). This creates file FORT1:IFORMAT.TEXT, which consists of SUBROUTINE IFORMT. Choose to **E**(XIT the Editor. *(Note: You could save yourself some typing by making sure the prefix is "FORT1:". I normally have the prefix set to the name of my disk, and it is for this reason that I am always specifying the FORT1 disk. Another option is to use "*", which stands for "boot diskette", as your disk name specifier.)*

b. From Command Level, choose once again to **E**(DIT. Now enter SUBROUTINE FFORMT. When finished, **Q**(UIT and **W**(RITE file "FORT1:FFORMAT". You now have SUBROUTINE FFORMT saved as FORT1:FFORMAT.TEXT. **E**(XIT the Editor.

c. Enter the **E**(ditor once again, type SUBROUTINE AFORMT and write it to disk as "FORT1:AFORMAT".

d. Enter SUBROUTINE EFORMT and write it to FORT1 as "FORT1:EFORMAT".

e. Finally, enter SUBROUTINE ADDRSS and write it as

"FORT1:ADDRESS". Make sure all of your subroutine names and disk file names are *exactly* the same as those listed here.

f. Now we will compile each of our library files. To do so, choose Command Level option **C**(OMPILE. We do *not* link library files until we have a Main Program, so do *not* choose to **R**(UN. Listed below are the Compiler prompts and the appropriate replies for the first two library files:

```
COMPILE WHAT TEXT ? FORT1:IFORMAT
TO WHAT CODEFILE ? $
LISTING FILE ? <RETURN>

COMPILE WHAT TEXT ? FORT1:FFORMAT
TO WHAT CODEFILE ? $
LISTING FILE ? <RETURN>
```

Compile all five library files.

g. At this point, the boot diskette contains a total of ten extra files (five Text & five Code library files). To make sure we have room for the Main Program, enter the **F**(ILER and **K**(RUNCH disk FORT1. You may also want to list its directory and see how *small* the code files are. (Remember, we didn't link them.)

h. Now it's time to enter the Main Program. Enter the Editor. *Before* typing the FORMAT program as shown in Chapter 8, type these lines *exactly* as shown, with the "$" in Column one:

```
$       USES UIFORMT IN FORT1:IFORMAT.CODE
$       USES UFFORMT IN FORT1:FFORMAT.CODE
$       USES UAFORMT IN FORT1:AFORMAT.CODE
$       USES UEFORMT IN FORT1:EFORMAT.CODE
$       USES UADDRSS IN FORT1:ADDRESS.CODE
```

You follow these lines with the program itself. When you finish, **Q**(UIT and **U**(PDATE THE WORKFILE. (Note that the Main Program *is* saved as the workfile.)

i. **C**(OMPILE the workfile. Again, do *not* choose the **R**(UN option! The "COMPILE WHAT TEXT ?" and "TO WHAT CODEFILE ?" prompts are answered automatically. Answer the "LISTING FILE ?" prompt appropriately and compiling will commence.

j. We now have six independently compiled programs: five library files, and one Main Program (or calling program). Let's call in our friend the Linker to finish this project. Here are your answers for his queries:

```
HOST FILE ? <RETURN>
LIB FILE ? *
LIB FILE ? FORT1:IFORMAT
LIB FILE ? FORT1:FFORMAT
LIB FILE ? FORT1:AFORMAT
LIB FILE ? FORT1:EFORMAT
LIB FILE ? FORT1:ADDRESS
LIB FILE ? <RETURN>
MAP FILE ? <RETURN>
OUTPUT FILE ? <RETURN>
```

k. You've done it! (*Reminder: "CODE WRITE ERR" means your disk was not properly Krunched, so there was no room*). X(ECUTE file "**SYSTEM.WRK**", and you'll see that the six individual pieces work as a whole.

l. (Optional) For a real kick, try adding the $XREF Compiler Directive to the Main Program, then choose to send the Listing File to the Console when you re-compile. (Be prepared to use CTL-S.) Then, when you re-link, send the Linker's Map File to the Console, also. (Type "**#1:**" when the "MAP FILE ?" prompt appears.) Call over a friend and show him the Linking process. (Hint: Try to look sophisticated and make him think you understand everything about the Map File. He'll be impressed, especially if you act bored—as if you've done this a thousand times.)

m. Don't forget to remove the clutter from FORT1 when you finish. Save the files on your own disk if you wish to have them preserved.

Chapter 16:
FORMATTED DATA FILES

A. FORMATTED SEQUENTIAL DATA FILES

To avoid confusion, we will refer to FORTRAN's Text (information) files as *Data files* (since *all* FORTRAN programs are Text files when originally created). FORTRAN supplies two Data file structures: *sequential* and *random-access* (or *direct-access,* as they are also called).

Here are some comparisons between FORTRAN's and Applesoft BASIC's DOS sequential Data files:

1. Both FORTRAN and BASIC sequential files contain one piece of information immediately after another.

2. BASIC sequential files consist of one field after another. FORTRAN sequential files consist of one record after another. These records in turn consist of fields.

3. BASIC sequential files contain no padding (i.e., room for expansion) between fields. FORTRAN files have no padding between records, but you may have padding between the fields that make up the record.

4. BASIC uses both the RETURN character (ASCII 13) and commas to separate fields. FORTRAN uses *only* the RETURN character to separate records (fields are not separated from adjacent fields at all).

5. It is not very easy to move the file pointer around in sequential files in either language. It is somewhat easier in FORTRAN, because you may jump one record at a time either forward *or* backward.

6. Locating the end of a FORTRAN sequential file is more straightforward than it is for a BASIC file, because FORTRAN has a special "END OF FILE" character that can be searched for (with an additional READ statement parameter).

1. Structure

Here is the structure of a FORTRAN Sequential file (<ret> signifies the RETURN character, ASCII 13):

```
fieldfieldfieldfield<ret>fieldfield<ret>fieldfieldfieldfieldfield<ret>

etc.
```

This attempts to show that the *fields* in a record are not separated. In addition, each *record* (the segment between RETURNs) may be of a different length and contain a different number of fields.

2. Commands

It is important when writing file development programs to know your position within the file at all times. With this in mind, those commands which affect position will clearly state that a file "pointer" has been moved.

The form of each command will be presented first, followed by an explanation of its function, and finally a few sample uses.

a. `OPEN (unit number, FILE = 'filename', STATUS = 'status')`

The OPEN statement *opens* a Sequential Data file, *associates* a program unit number (or *internal file*) with an operating system diskfile (or *external file*), and *moves the file pointer to the beginning of the file.*

"Unit number" is an Integer expression. (Note: "0" is reserved for "CONSOLE:", however.) This Integer will then be used in all READs and WRITEs to the file.

"Filename" is the name of the diskfile. I recommend using a disk name even if the file will be written to FORT1, for many users elect to change the default disk prefix.

"Status" is one of two words:

"NEW" is used *only when you are creating the file.* "NEW" will wipe out any file already existing under the current filename!

"OLD" is used when you are reading from or writing to a *previously created* file. This is the *default* status. In other words, one need not even specify the STATUS option if the status is "OLD".

Here are some sample OPENs. Can you interpret them?

```
OPEN (4, FILE = 'TELE:PHONE.DATA', STATUS = 'NEW')
OPEN (5, FILE = 'BLACK:BOOK.DATA')
OPEN (4, FILE = 'BAD:WEATHER.DATA', STATUS = 'NEW')
```

b. `READ (unit, format line number, END = line number) variable(s)`

This is the familiar READ statement. *READ advances the file pointer one record* (from one RETURN character to the next). In other words, READ reads an entire record at once. This *record* may contain one or more *fields.*

"unit" will now be the unit number given in the OPEN statement (instead of "*").

"END = line number" is a new option for use in reading from Data files. It prevents one from reading past the end of the file, and as such is a very valuable parameter. When the special "end of file" character is read, program control will branch to the specified line number instead of generating an error message. This is a quick and safe way to find the end of the file when processing each file record sequentially, or when appending a file.

Some sample READs:

```
READ (4, 2ØØ, END = 25Ø) PHONE
READ (5, 1Ø, END = 5Ø) ADDRSS
READ (4, 1ØØ, END = 15Ø) DATE, HI, LO
```

c. WRITE (unit, format line no.) output list

This is the familiar WRITE statement. *WRITE moves the file pointer ahead one record.* In other words, WRITE always writes a *complete* record at one time, beginning at the first byte of the record. (There is no command analogous to a BYTE parameter.) This *record* may of course contain one or more *fields.*

Some sample WRITEs:

```
WRITE (4, 5Ø) NAME (INDEX), ADDRSS (INDEX)

WRITE (5, 1ØØ) NAME, GRADE
```

d. FORMAT (format specifiers)

When coupled with a WRITE statement, a FORMAT statement determines the make-up of each individual file record, since *one FORMAT governs the structure of precisely one record.* FORMAT sends the RETURN character automatically at the end of the data to serve as a record separator. Let's look at two examples:

```
        WRITE (4, 1Ø) NAME, ADDRSS
1Ø      FORMAT (A2Ø, A15)
```

This WRITE-FORMAT pair will create one *record* consisting of two *fields* (a 20-character field and a 15-character field). The record will be written

from the file pointer's current position forward. After the record has been written, the pointer will be at the *end* of the record.

```
        WRITE (5, 2Ø) DATE, HIGH, LOW
2Ø      FORMAT (A8, I3, I3)
```

Now a three-field record is written to the file opened with unit number 5.

Technical aside: It is possible to write *more* than one record with a single FORMAT if one uses the "/" character. Recall that "/" means "go to the next line," since it generates the RETURN character. We now extend it to mean "go to the next record." Likewise, it is possible to write a *portion* of a record with a single FORMAT by using the "$" character. Recall that "$" means "stay on the same line," since it suppresses the automatic RETURN at the end of a FORMAT. We now extend this to mean "stay in the same record," or "don't send the RETURN character to end the record yet."

NOTE: Since the FORMAT statement determines record structure when the file is being written, you must use precisely the same FORMAT to read the records.

By now, you should see a parallel between writing lines to the Console and writing records in a file. That is because a line of output on the Console *is* a record! *The only difference between writing a record (or line) on the Console and writing a record to a file is the destination of that record. (One goes to the Console, the other goes to a file.)*

e. ENDFILE unit number

The ENDFILE statement generates the special "end of file" character CTL-C (ASCII 3) at the file pointer's current position. *This should always be written as the last record of a file.* This statement does *not* have to be included in a WRITE (it is an independent statement). Here's how it might look:

```
ENDFILE 4
```

NOTE: The file pointer is positioned after the "EOF" character, once it has been written.

f. BACKSPACE unit number

BACKSPACE does exactly what its name implies: *it backs up the file pointer one record.* There are two reasons for this statement:

First, with it one may move the file pointer backwards through a file. Second (and more important), it allows one to position the file pointer *before* the "end of file" character when searching for it in order to append a file.

Recall the "END =" option of READ. Also recall that READ moves the file pointer one record forward. *Where is the pointer after it reads the "end of file"? The record* after *the "end of file." Attempting to READ or WRITE will now cause a Run-time error unless you first BACKSPACE.* In the same way, one may change the contents of a record by searching for it with READ; after finding the selected record, one may backspace to the beginning of the record, then write the new contents *over* the old.

A typical backspace looks like this:

```
BACKSPACE 5
```

NOTE: The above does not mean backspace 5 records. It means back up one record on unit 5.

g. `REWIND unit number`

It isn't difficult to figure out the origin of this statement. *REWIND moves the file pointer to the first record of the file,* which in earlier days literally meant rewinding a tape.

NOTE: If you repeatedly search through a file (for a name, or date, etc.), you must REWIND the file between each search.

Remember that "OPEN" (discussed earlier) has "REWIND" built into it.

Here is a typical REWIND:

```
REWIND 4
```

h. `CLOSE (unit number, STATUS = 'status')`

"status" is one of two options:

"DELETE" will delete *only files opened as 'NEW'* (newly created). Why would anyone want to delete a newly created file? Usually this is done when you are debugging a file development program, when you create a "trash" file to see if the program works correctly. Since these trash files are merely experimental, there is no reason to save them. The DELETE option prevents an unnecessary trip to the Filer.

"KEEP" allows the file to remain on the disk. This is the *default* status.

The CLOSE command closes a file. It is wise to close every opened file to prevent unwanted data from being added to the file accidentally. (Technical

aside: It also forces an internal file buffer to write its contents to the external disk file.)

Following are a few examples of CLOSE:

```
CLOSE (4)

CLOSE (5)

CLOSE (4, STATUS = 'DELETE')
```

3. Sample Program

We will now run through a simple program to create and read a file containing two records. Let's look at it in its entirety, and then examine it line by line:

```
        PROGRAM EXAMPL
        INTEGER DAY, YEAR
        REAL SECNDS
        CHARACTER MONTH*3

C       Write two records
        OPEN (4, FILE = 'RUNNER:TIME.DATA', STATUS = 'NEW')
        WRITE (4, 1Ø) 'Jul', 7, 84, 9, 48.3
1Ø      FORMAT (A3, I2, I2, I2, F5.2)
        WRITE (4, 1Ø) 'Jul', 9, 84, 1Ø, 8.368
        ENDFILE 4

        REWIND 4

C       Read and print records until <eof>
2Ø      READ (4, 1Ø, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS
C       If <eof> has not been read then
        WRITE (*, 1Ø) MONTH, DAY, YEAR, MINITS, SECNDS
        GOTO 2Ø

C       If <eof> has been reached then
3Ø      CLOSE (4, STATUS = 'DELETE')
        END
```

We will now examine the program in segments. The segments are followed in **emphasized print** by the file as it would appear *after* the execution of the statements.

Special characters:
"∧" represents the file pointer
"ϕ" represents the space character (ASCII 32)
"<ret>" represents the RETURN character (ASCII 13)
"<eof>" represents the End Of File character (ASCII 3)

```
PROGRAM EXAMPL
INTEGER DAY, YEAR
REAL SECNDS
CHARACTER MONTH*3
OPEN (4, FILE = 'RUNNER:TIME.DATA', STATUS = 'NEW')

^
```

Comment: The file pointer is moved to the beginning of a new file.

```
        WRITE (4, 1Ø) 'Jul', 7, 84, 9, 48.3
1Ø      FORMAT (A3, I2, I2, I2, F5.2)

        Julϕ784ϕ948.3Ø<ret>
                        ^
```

```
     WRITE (*, 1Ø) 'Jul', 7, 84, 9, 48.3
1Ø   FORMAT (A3, I2, I2, I2, F5.2)
```

Fields 1 – 5

```
AAA II II II FF.FF <ret>
Jul ϕ7 84 ϕ9 48.3Ø <ret>
```

The Format specifiers have total control over field width & type (Character, Integer, or Real). The output list items are merely fit into the record as prescribed by the FORMAT.

Figure 16.1 How a FORMAT Determines Record Structure

Comments: Note how values are "fit" into the specified FORMAT just as they are when you are writing to the Console; also notice that all five fields are adjacent; the <ret> is sent automatically by the FORMAT.

```
WRITE (4, 1Ø) 'Jul', 9, 84, 1Ø, 8.368

Julϕ784ϕ948.3Ø<ret>Julϕ9841Øϕ8.37<ret>
                                    ^
```

Comment: The file pointer continues to move along as records are written.

```
ENDFILE 4

Jul⊥784⊥948.3Ø<ret>Jul⊥9841Ø⊥8.37<ret><eof>
                                          ^
```

Comment: Notice that the file pointer is *past* <eof>

```
REWIND 4

Jul⊥784⊥948.3Ø<ret>Jul⊥9481Ø⊥8.37<ret><eof>
^
```

Comment: The file pointer is moved back to the beginning of the file so records may be read; the next portion of program reads each record and prints it on the Console until <eof> is reached; the FORMAT for the Console is exactly the same as that for the file, so you can see on the screen what the file really looks like.

```
2Ø      READ (4, 1Ø, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS
C       If <eof> has not been read then
        WRITE (*, 1Ø) MONTH, DAY, YEAR, MINITS, SECNDS
        GOTO 2Ø
```

Comment: Following are the three iterations of the READ statement from the above loop:

```
2Ø      READ (4, 1Ø, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS

        Jul⊥784⊥948.3Ø<ret>Jul⊥9841Ø⊥8.27<ret><eof>
                          ^
```

Comments: READ reads to <ret>; note that the same FORMAT is used for WRITE *and* READ; values of variables after first iteration — MONTH = 'Jul', DAY = 7, YEAR = 84, MINITS = 9, SECNDS = 48.3

```
2Ø      READ (4, 1Ø, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS

        Jul⊥784⊥948.3Ø<ret>Jul⊥9841Ø⊥8.37<ret><eof>
                                         ^
```

Comment: Values of variables after second iteration — MONTH = 'Jul', DAY = 9, YEAR = 84, MINITS = 10, SECNDS = 8.37

```
2Ø      READ (4, 1Ø, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS

        Jul⊥784⊥948.3Ø<ret>Jul⊥9841Ø⊥8.37<ret><eof>
                                          ^
```

Comment: The third iteration of line 20 reads <eof>, so program control branches to line 30:

```
C       If <eof> has been reached then
3Ø      CLOSE (4, STATUS = 'DELETE')
        END
```

Comment: The file is closed and the program ends.

4. Blurring the Distinction Between Sequential and Direct-Access Files

Recall that *Sequential* files have information packed with no extra padding, and may contain records of varying lengths.

Direct-access files pad each record to a specified length, which allows easy movement from one record to any other record in the file.

The following list enumerates how the differences between the two file structures can become insignificant in FORTRAN:

a. We have already seen how file pointer movement *is* possible in Sequential files by using BACKSPACE (and REWIND) to move backward, and READ to move forward. By using these commands in loops, we may move any number of records we choose either forward or backward.

b. While it is possible to create files with different record lengths (i.e., by using different FORMAT statements), Sequential files are most often created by repeatedly using the *same* FORMAT statement. When one does so, all records will naturally be of the same length.

c. Finally, even the question of padding to allow future expansion breaks down in FORTRAN. If one wishes to make each record 100 characters long, but has only two fields to store currently, he or she may use a FORMAT similar to the following:

```
FORMAT (A5, I5, 9ØX)
```

So the only real distinction between Sequential and Direct-access files in FORTRAN is that *Direct-access files allow one to move directly from one record to another without having to pass over every record in between.* Since this difference is so trivial, we will *not* discuss Formatted Direct-access files in detail.

B. FORMATTED DIRECT-ACCESS DATA FILES (optional material)

Following is a brief description of the commands used with Direct-access files.

1. `OPEN(u,FILE='filename',STATUS='status',ACCESS='DIRECT',RECL=length)`

Notice the addition of the "ACCESS" and "RECL" (record length) parameters. Record length should be the sum of all field widths, with consideration for future expansion. Remember, there is nothing separating fields, so you need not include anything extra for field separators. Here is a typical OPEN:

```
OPEN (4, FILE='POLICE:FINE.DATA', STATUS='NEW', ACCESS='DIRECT', RECL=5Ø)
```

2. READ (u, format line, REC=record no., END=line number) variable(s)

Note the "REC" parameter, which serves to position the file pointer at the specified record *before* READing. Sample (note the use of a variable for the record number):

```
READ (4, 1Ø, REC = RECORD, END = 5Ø) SCHOOL, CITY
```

NOTE: The first record of a file is number 1 (A record 0 does not exist.)

3. WRITE (unit, format line number, REC = record number) output list

Again note the "REC = " parameter to position the file pointer. Sample use:

```
WRITE (5, 2Ø, REC = COUNTR) WORKER, HOURS
```

4. FORMAT (format specifiers)

Identical to Sequential file FORMATs, except that the FORMAT is no longer the precise record "blueprint." It is likely that after the data are sent to the file according to the FORMAT, additional padding will be needed (to fill out the record length).

5. ENDFILE unit, BACKSPACE unit, REWIND unit, CLOSE (unit, STATUS = 'status')

These are identical to the Sequential file commands (although BACKSPACE and REWIND are redundant for Direct-access files since the file pointer is positioned for every READ and WRITE anyway).

C. REVIEW QUESTIONS

1. Describe the format of a Formatted Sequential file.

2. Which Sequential file commands move the file pointer forward? Which commands move it backward?

3. If unit 0 represents the Console, then it would be allowable for us to use "WRITE (0, 10) . . ." synonymously with "WRITE (*, 10) . . .". I feel the "*" is *much* more worthwhile (note that I have used it throughout the book). Can you guess my reason?

4. Would it be possible to open a file on a given *drive,* rather than on a given *disk?*

5. I feel that 4 and 5 are the best unit numbers to use for opening disk files, while 6 is best for opening the Printer as a file. Do you know why I feel this way?

D. EXERCISES

1. This program will allow you to create your own file with names and phone exchanges, then move forward and backward through the file, and finally determine which record the file pointer is on with a "READ CURRENT RECORD" option. The program uses the STATUS = 'DELETE' option of CLOSE, so your file will not be saved on disk.

WARNING: This program has no error trapping other than telling you that you've reached the "end of file" character. It is very likely that you will get Run-time errors the first few times you execute the program (you read past "end of file," you forget to REWIND, etc.). The purpose is, of course, to learn from your mistakes.

```
      PROGRAM FILE
      INTEGER CHOICE, RINGS
      CHARACTER HOME*1, INVRSE*1, NORMAL*1, BELL*1, DELAY*1, NAME*3Ø

      HOME = CHAR (12)
      INVRSE = CHAR (15)
      NORMAL = CHAR (14)
      BELL = CHAR (7)
      ASSIGN 3Ø TO MENU

C     Title page
      WRITE (*, 5) HOME
```

```
5       FORMAT (A, $)
        WRITE (*, 1Ø) 'Welcome to'
1Ø      FORMAT (/, /, /, 23X, A, /)
        WRITE (*, 2Ø) INVRSE,
     *     '*** LEARNING APPLE FORTRAN SEQUENTIAL FILES ***', NORMAL
2Ø      FORMAT (5X, A, A, A, /, /, /)

        WRITE (*, 25)
     *     'Note: The data file will be temporarily saved on the disk'
25      FORMAT (A)
        WRITE (*, 25)
     *     'in Drive 1. It will be deleted at the program''s end.'

3Ø      WRITE (*, 4Ø) 'Press RETURN to continue...'
4Ø      FORMAT (/, /, /, 14X, A, $)
        READ (*, 25) DELAY

C       Menu Page
45      WRITE (*, 5) HOME
        WRITE (*, 5Ø) INVRSE, 'Available options', NORMAL
5Ø      FORMAT (10X, A, A, A, /)

        WRITE (*, 6Ø) '1. Create a new sequential file'
6Ø      FORMAT (7X, A)
        WRITE (*, 6Ø) '2. Read the current record'
        WRITE (*, 6Ø) '3. Append the file'
        WRITE (*, 6Ø) '4. Rewind the file'
        WRITE (*, 6Ø) '5. Backspace'
        WRITE (*, 6Ø) '6. Move forward'
        WRITE (*, 6Ø) '7. End'

        WRITE (*, 7Ø) 'Your choice (1 - 7) ? '
7Ø      FORMAT (/, /, 1ØX, A, $)
        READ (*, 8Ø) CHOICE
8Ø      FORMAT (I1)

        WRITE (*, 5) HOME
        GOTO (9Ø, 1ØØ, 11Ø, 12Ø, 13Ø, 14Ø, 15Ø) CHOICE

C       Otherwise invalid reply

        DO 85 RINGS = 1, 5
           WRITE (*, 5) BELL
```

```
85      CONTINUE

        WRITE (*, 25) 'Invalid option!'
        GOTO MENU

9Ø      CALL CREATE
        GOTO MENU

1ØØ     CALL READ
        GOTO MENU

11Ø     CALL APPEND
        GOTO MENU

12Ø     CALL REWIND
        GOTO MENU

13Ø     CALL BCKSPC
        GOTO MENU

14Ø     CALL FORWRD
        GOTO MENU

15Ø     CLOSE (4, STATUS = 'DELETE')
        END

        SUBROUTINE CREATE
C       Creates formatted sequential file

        INTEGER EXCHNG
        CHARACTER NAME*3Ø

        OPEN (4, FILE = '#4:PHONE.DATA', STATUS = 'NEW')

        WRITE (*, 1Ø) 'Creates a Formatted Sequential data file.'
1Ø      FORMAT (A, /)
        WRITE (*, 2Ø) 'Each record will consist of a 3Ø byte name'
2Ø      FORMAT (A)
        WRITE (*, 2Ø) 'and a 3 digit phone exchange'

3Ø      WRITE (*, 4Ø) 'Enter name (format A3Ø, "END" to escape) : '
4Ø      FORMAT (/, A, $)
        READ (*, 5Ø) NAME
5Ø      FORMAT (A3Ø)
```

```
      IF (NAME .NE. 'END') THEN

         WRITE (*, 4Ø) 'Enter 3 digit phone exchange : '
         READ (*, 6Ø) EXCHNG
6Ø       FORMAT (I3)

C        Write record in file
         WRITE (4, 7Ø) NAME, EXCHNG
7Ø       FORMAT (A3Ø, I3)
         GOTO 3Ø

      ELSE

C        Place end of file character
         ENDFILE 4

      ENDIF

      END

      SUBROUTINE READ
C     Reads current record

      INTEGER EXCHNG
      CHARACTER NAME*3Ø

      READ (4, 1Ø, END = 3Ø) NAME, EXCHNG
1Ø    FORMAT (A3Ø, I3)

C     If end of file has not been reached, then
      WRITE (*, 2Ø) 'The current record is :'
2Ø    FORMAT (A, /)
      WRITE (*, 1Ø) NAME, EXCHNG
      GOTO 4Ø

C     Otherwise
3Ø    WRITE (*, 2Ø) 'You are at the end of the file...'

4Ø    END

      SUBROUTINE APPEND
C     Adds a record to the file's end

      INTEGER EXCHNG, RECORD
      CHARACTER NAME*3Ø
```

```
C       Read to the end of the file
        DO 2Ø RECORD = 1, 1ØØØ
           READ (4, 1Ø, END = 3Ø) NAME, EXCHNG
1Ø         FORMAT (A3Ø, I3)
2Ø      CONTINUE

3Ø      BACKSPACE 4

C       Accept new record
        WRITE (*, 4Ø) 'Enter name to add to file : '
4Ø      FORMAT (/, A, $)
        READ (*, 5Ø) NAME
5Ø      FORMAT (A3Ø)
        WRITE (*, 4Ø) 'Enter 3 digit exchange : '
        READ (*, 6Ø) EXCHNG
6Ø      FORMAT (I3)

C       Write new name in file and add new end-of-file character
        WRITE (4, 1Ø) NAME, EXCHNG
        ENDFILE 4

        END

        SUBROUTINE REWIND
C       Moves file pointer to beginning of file

        REWIND 4
        WRITE (*, 1Ø) 'File has been rewound...'
1Ø      FORMAT (A)

        END

        SUBROUTINE BCKSPC
C       Moves file pointer backwards

        INTEGER BACKUP, RECORD

        WRITE (*, 1Ø) 'Backspace how many records (format I2) ? '
1Ø      FORMAT (A, $)
        READ (*, 2Ø) BACKUP
2Ø      FORMAT (I2)

        DO 3Ø RECORD = 1, BACKUP
           BACKSPACE 4
3Ø      CONTINUE
```

```
      WRITE (*, 4Ø) 'Backspacing completed. . .'
4Ø    FORMAT (/, A)

      END

      SUBROUTINE FORWRD
C     Moves file pointer forward

      INTEGER EXCHNG, AHEAD, RECORD
      CHARACTER NAME*3Ø

      WRITE (*, 1Ø) 'Move forward how many records (format I2) ? '
1Ø    FORMAT (A, $)
      READ (*, 2Ø) AHEAD
2Ø    FORMAT (I2)

      DO 4Ø RECORD = 1, AHEAD
         READ (4, 3Ø) NAME, EXCHNG
3Ø       FORMAT (A3Ø, I3)
4Ø    CONTINUE

      END
```

2. Recall the grade-point calculation program given at the end of Chapter 12. The program user entered a student name, followed by a list of course titles, instructors, credit hours, and letter grades for each course in which the student was enrolled. Modify this program as follows:

a. After accepting the information for a student, write it to a Formatted Sequential file. Then *continue to accept information for other students,* writing each set of student data as one record. For the sake of simplicity, assume each student has enrolled in five courses. Each record will therefore contain twenty-one fields: the student name and five of each of the following: course name, instructor, credit hours, and letter grade.

b. Once the file has been created, allow the program user to report either all of the students contained in the file, or else report only a single student (this name must be specified, of course).

You may wish to add options to append the student file with more records, and to edit an incorrect student record. Print all program user prompts and menus on the Console, but have the grade reports appear on the Printer.

Chapter 17:
UNFORMATTED DATA FILES

A. UNFORMATTED SEQUENTIAL DATA FILES

1. Introduction

FORTRAN provides an extremely efficient Data file storage system known as *Unformatted files.* Unformatted files have absolutely no system "overhead" to slow their implementation. Normally, file input and output are subject to system interpretation. (Think of how many different ways the number "4532" could be read depending on the given FORMAT, for example.) The use of Unformatted files involves no attempt to READ or WRITE a number as an F6.2 number, or an F4.1, or an I3, etc., but rather involves READing or WRITEing the number (or Character value) *as is (i.e., in its pure BINARY form).*

There are several advantages to using Unformatted files:

a. They are faster in use than Formatted files (anywhere from roughly two to ten times faster, depending on the file length).

b. They require less storage space (roughly half that of a typical Formatted file, again depending upon the application).

c. They are written with the exact same commands as Formatted files with one minor exception: *READs and WRITEs to an Unformatted file require no corresponding FORMAT statements.*

There are also a few disadvantages of using Unformatted *Sequential* files:

a. The file pointer may not be moved backwards one record at a time. (You will soon see that there *are* no records in an Unformatted Sequential file.) You must always REWIND to the *beginning* of the file.

b. Unformatted Sequential files allow no "padding." Therefore, one may not use this file structure when it is necessary to update the file (unless the update is added to the *end* of the file, not to the middle).

These disadvantages do *not* apply to Unformatted *Direct-access* files, which we will cover later in this lesson.

2. Structure

An Unformatted Sequential file may be represented as follows:

```
fieldfieldfieldfieldfieldfield etc.
```

Note that fields are adjacent, and that *there is no such file unit as a record.* This accounts for the disadvantages of Unformatted file use listed above. If you're inquisitive, you may wonder how the fields can be separated if there is no system interpretation (FORMATs) when you are using Unformatted files. The answer is that the *type* of variable being written or read determines field width, as we shall soon see.

3. Commands

Following is a list of the commands available when using Unformatted Sequential files.

a. `OPEN (unit, FILE='filename', STATUS='status', FORM='UNFORMATTED')`

The "unit", "FILE", and "STATUS" ('NEW' or the default 'OLD', remember?) options are identical to what we discussed in the last chapter on Formatted files. The only thing different is the addition of the Unformatted FORM designation. This in effect tells the operating system to keep its hands off the contents being sent to and from the files.

NOTE: "OPEN" moves the file pointer to the beginning of the specified file.

Here are a few sample OPENs:

```
OPEN (4, FILE = 'BANK:CHECK.DATA', STATUS = 'NEW', FORM = 'UNFORMATTED')
OPEN (5, FILE = 'OFFICE:SOCSEC.DATA', FORM = 'UNFORMATTED')
```

b. `READ (unit, END = line number) variable(s)`

Note that the only difference from the Formatted file READ is that *no FORMAT is specified.* The variables you list are read uninterpreted from the file opened with the given unit number. Here are some examples:

```
READ (4, END = 25) NAME, ADDRSS

READ (5, END = 1ØØ) ZIP

READ (4, END = 3Ø) NAME (INDEX), GRADE (INDEX)
```

NOTE: "READ" MOVES THE FILE POINTER AHEAD ONE FIELD AT A TIME PER VARIABLE LISTED. How will READ know when a field ends if there are no field separators? By the type of variable it is reading! An INTEGER variable in READ will advance the file pointer TWO bytes, a REAL

variable moves it FOUR bytes, and a CHARACTER variable the DECLARED CHARACTER VARIABLE LENGTH.

c. WRITE (unit) output list

Notice again the absence of a Format. The values listed in WRITE are sent unaltered (in binary form) to the specified file. They are placed at the current file pointer's position. *The file pointer then advances with each field written.* How might an Unformatted WRITE look?

```
WRITE (4) NAME, SOCSEC

WRITE (5) NAME (PERSON), CITY (PERSON), STATE (PERSON)
```

d. ENDFILE unit number

This is identical to the ENDFILE used with Formatted files: it places the special "end of file" character at the file pointer's current position, to prevent one from reading past it. This is picked up by the "END = " option of READ.

e. REWIND unit number

Moves the file pointer back to the beginning of the file. Don't forget to do so every time you search through your file, or you'll read past the end of the file (a Run-time error).

f. CLOSE (unit number, STATUS = 'status')

As are most of the commands for Unformatted files, this is identical to its Formatted file equivalent.

g. Illegal commands

1. FORMAT—for obvious reasons.
2. BACKSPACE—since Unformatted Sequential files *have* no record structure, it is impossible to move the pointer back one record.

h. Sample program

We will now examine the same sample program as we did in the previous chapter, except that the data will now be written in an Unformatted Sequential file.

```
      PROGRAM EXAMPL
      INTEGER DAY, YEAR
      REAL SECNDS
      CHARACTER MONTH*3

C     Write two unformatted records
      OPEN (4, FILE = 'RUNNER:TIME.DATA', STATUS = 'NEW',
     *      FORM = 'UNFORMATTED')
      WRITE (4) 'Jul', 7, 84, 9, 48.3
      WRITE (4) 'Jul', 9, 84, 10, 8.368
      ENDFILE 4

      REWIND 4

C     Read and print each record until <eof>
10    READ (4, END = 30) MONTH, DAY, YEAR, MINITS, SECNDS
C     If <eof> has not been read then
      WRITE (*, 20) MONTH, DAY, YEAR, MINITS, SECNDS
20    FORMAT (A3, 1X, I2, 1X, I2, 1X, I2, 1X, F5.2)
      GOTO 10

C     If <eof> has been reached then
30    CLOSE (4, STATUS = 'DELETE')
      END
```

As before, we will review the program in segments. Each command which affects the file will be followed by a representation of the file in emphasized print. ***(Note: For the sake of simplicity and readability, the file values will be represented in their normal, base 10 forms. In reality, they would be written in binary form.)***

```
      PROGRAM EXAMPL
      INTEGER DAY, YEAR
      REAL SECNDS
      CHARACTER MONTH*3
      OPEN (4, FILE = 'RUNNER:TIME.DATA', STATUS = 'NEW',
     *      FORM = 'UNFORMATTED')

      ^
```

Comment: The file pointer is moved to the beginning of a new file.

```
WRITE (4) 'Jul', 7, 84, 9, 48.3

Jul784948.3
          ^
```

Comments: Notice that all five fields are adjacent; no FORMAT is used to write the data, so there are no blank spaces, and no added zeroes; no <ret> is sent to file, either. This lack of system-supplied "extras" is what accounts for an Unformatted file's space efficiency.

```
WRITE (4) 'Jul', 9, 84, 1Ø, 8.368

Jul784948.3Jul9841Ø8.368
                       ^
```

Comment: The file pointer continues to move along as fields are written.

```
ENDFILE 4

Jul784948.3Jul9841Ø8.368<eof>
                         ^
```

Comment: Notice that the file pointer is *past* <eof>.

```
REWIND 4

Jul784948.3Jul9841Ø8.368<eof>
^
```

Comments: The file pointer is moved back to the beginning of the file so records may be read; the next portion of program reads each record and prints it on the Console until <eof> is reached.

```
1Ø      READ (4, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS
C       If <eof> has not been read then
        WRITE (*, 2Ø) MONTH, DAY, YEAR, MINITS, SECNDS
2Ø      FORMAT (A3, 1X, I2, 1X, I2, 1X, I2, 1X, F5.2)
        GOTO 1Ø
```

Comments: *Even though the data were written and read in Unformatted form, you MUST use a FORMAT to write the data on the Console (the Console is said to be a FORMATTED DEVICE).* Following are the three iterations of the **READ** statement from the above loop:

```
1Ø      READ (4, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS

        Jul784948.3Jul9841Ø8.368<eof>
                  ^
```

Comments: The file pointer moved three bytes for Character*3 variable MONTH; two bytes each for Integer variables DAY, YEAR, and MINITS;

and four bytes for Real variable SECNDS (again, remember that the data are actually stored in binary form).

```
1Ø      READ (4, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS

        Jul784948.3Jul9841Ø8.368<eof>
                                 ^
```

Comment: Values of variables after second iteration — MONTH = 'Jul', DAY = 9, YEAR = 84, MINITS = 10, SECNDS = 8.368

```
1Ø      READ (4, END = 3Ø) MONTH, DAY, YEAR, MINITS, SECNDS

        Jul784948.3Jul9841Ø8.368<eof>
                                 ^
```

Comment: The third iteration of Line 10 reads <eof>, so program control branches to Line 30.

```
C       If <eof> has been reached then
3Ø      CLOSE (4, STATUS = 'DELETE')
        END
```

Comment: The file is closed and the program ends.

4. Review Question

List the major differences in structure and commands between Formatted and Unformatted Sequential files.

5. Exercise

Write a program which will accept a list of transactions for your savings account. The program should first ask its user to enter a transaction code (D(eposit, W(ithdrawal, or Q(uit), then the amount of cash deposited or withdrawn. These two fields (the code and the cash) should then be saved in an Unformatted Sequential file. Repeat the transaction prompt until the program user types Q(uit. At this point, generate a table to report the transactions, cash, and balance after each transaction (assume a beginning balance of $0). To do so, you will have to go to the beginning of the file and retrieve each field until you reach the end of the file. Note that it is not necessary to save the balance in the file.

Those desiring a more challenging program may ask for the date of the transaction. This may be saved in the file, also. When the month changes, have your report calculate interest.

B. UNFORMATTED DIRECT-ACCESS DATA FILES

1. Structure

The *ultimate* file structure in Apple FORTRAN is Unformatted Direct-access. It combines the best of both possible worlds: the ease of pointer movement and expansion given by Direct-access files, and the speed and space efficiency of Unformatted files.

Here is the structure of an Unformatted Direct-access file:

```
fieldfieldfield...padding...fieldfieldfield...padding..., etc.
```

Fields are adjacent and records are padded to a predetermined length. (This allows movement in one-record increments, since all records have the same length.)

2. Commands

a.
```
OPEN (unit, FILE='name', STATUS='status', FORM='UNFORMATTED',
*         ACCESS='DIRECT',RECL=length)
```

There are quite a few parameters that must be specified! Luckily, STATUS, FORM, ACCESS, and RECL may be given in any order.

To determine record length, recall that Integers require two bytes of storage. (Technical aside: Now we're storing *values,* not *digits*; the *value* "32767" can be stored in two bytes — 16 binary digits — of memory, although five bytes would be required to store the *digits* "32767.") Reals require four bytes, while Characters require one byte per character. It is *not* necessary to add anything extra for field separators (there *are* no field separators).

Let's set the record length for the following data base: A teacher gives four tests per quarter, and wants to save these four scores plus their average in a record for each of his students.

If we map the record structure, it will look like this:

```
Field 1 : student name (Character*25)
Field 2 : test average (Real)
Field 3 : number of chapters recorded (Integer)
Field 4 : test 1 (Integer)
Field 5 : test 2 (Integer)
Field 6 : test 3 (Integer)
Field 7 : test 4 (Integer)
```

Our record length is 25 + 4 + 2 + 2 + 2 + 2 + 2 = 39.

b.
```
READ (unit, REC = record, END = line number) variable(s)
```

The "REC = " parameter positions the file pointer before reading commences.

NOTE: Although the "END = " option is a valid parameter, I know from experience that it does not work with Unformatted Direct-access files. To prevent reading past the end of such a file, you must write a FLAG VALUE (such as "END" or "STOP") in your last record when creating or appending the file. You may then check for this flag each time you read a new record.

The following statement

```
READ (4, REC = 8) PLAYER, NUMBER, AVERGE
```

will be able to separate the adjacent fields in record 8 by reading the prescribed number of bytes for Character variable PLAYER, two bytes for Integer variable NUMBER, and four bytes for Real variable AVERGE.

NOTE: The first record of a file is numbered 1. (There is no record 0.)

c. `WRITE (unit, REC = record) output list`

This will write one complete record, including padding if necessary, after moving the file pointer to the specified record.

d. `ENDFILE unit number`

As noted above, you may use the "eof" character, but this file structure will *not* pick it up.

e. `REWIND unit, BACKSPACE unit, CLOSE (unit, STATUS = 'status')`

BACKSPACE and REWIND have little if any value, since file pointer movement is done at will in Unformatted Direct-access files.

3. Sample Subroutine

To help summarize, let's incorporate the new commands by writing a subroutine to create the student test average files as described above in section 2, letter a. It will accept a name for each student, then write it (with the other six fields set to zero) as a record in an Unformatted Direct-access file. Note that the last name written in the file will be the flag "END":

```
      SUBROUTINE CREATE
C     Creates student test score file
```

```
      INTEGER RECORD, CHPTRS, SCORE
      CHARACTER HOME*1, NAME*25
      DIMENSION SCORE(4)

C     Set all four test scores to zero
      DATA SCORE / 0, 0, 0, 0 /
C     Set test average and number of chapters to zero
      DATA AVERGE, CHPTRS, RECORD / 0.0, 0, 0 /

      OPEN (5, FILE = 'TEACHER:TEST.DATA', STATUS = 'NEW',
     *      FORM = 'UNFORMATTED', ACCESS = 'DIRECT', RECL = 39)

      HOME = CHAR (12)

      WRITE (*, 10) HOME
10    FORMAT (A, $)

C     Increment record counter and accept student name
15    RECORD = RECORD + 1
      WRITE (*, 20) 'Student name (A25, "END" will escape) ? '
20    FORMAT (/, A, $)
      READ (*, 30) NAME
30    FORMAT (A25)

C     Write record in file
      WRITE (5, REC = RECORD) NAME, AVERGE, CHPTRS, SCORE(1),
     *                        SCORE(2), SCORE(3), SCORE(4)

      IF (NAME .NE. 'END') THEN
C        Get next student
         GOTO 15
      ELSE
C        Close file
         ENDFILE 5
         CLOSE (5)
      ENDIF

      END
```

4. Review Questions

1. How can the fields of an Unformatted Direct-access record be read if there are no field separators?

2. A person wants to write a data base with the following record structure:

```
Field 1 : month (Character*3)
Field 2 : day (Integer)
Field 3 : high temperature (Integer)
Field 4 : low temperature (Integer)
Field 5 : precipitation (Real)
```

What record length will be required?

5. Exercises

1. Write a program to accept a high temperature, low temperature, a precipitation code ("R" for rain, "S" for snow, "N" for none), and precipitation measured (in inches) for each of a series of days. Place the data for each day in one record of an Unformatted Direct-access file. After all the data have been entered, give the program user four menu options: report the data for a single record (ask the program user which record to report), report the entire file (read each file record, then report average high and low temperatures, as well as total inches of rain and snow), append the file (add records to the end of the file), and a "quit the program" option.

2. Complete the test average data base described in the text by writing subroutines to update the file (to be used after each of the four tests), report the entire file (print each student's name, test scores, and average), edit any incorrect test scores or misspelled names, and append the file with new students. These four options, along with the create file option given in the text, should then be tied together with a Main Program which will consist of a menu and calls to the appropriate subroutines.

An excellent opportunity exists if you are in a classroom situation: each subroutine may be written by a different person (or group of persons). The subroutines may be compiled separately, then linked upon completion to form a working whole. This simulates in a very realistic way the manner in which businesses divide a large project into smaller modules, which are then assigned to different programmers and developed individually.

Chapter 18:
THE APPLESTUFF LIBRARY UNIT

A. CORRECTING A BUG

Apple FORTRAN's System Library contains units other than MAINSEGX and RTUNIT. The subject of this chapter is the APPLESTUFF unit, which contains subroutines for generating random numbers, sound, and reading the game paddles and their associated buttons. (Note: The game paddles and buttons will not be discussed until Chapter 21, however.)

NOTE: Because of an omission in the FORTRAN system disks, it is not possible to access the "APPLESTUFF" or "TURTLEGRAPHICS" library routines without first adding a Pascal file.

Use the Filer to list disk FORT1's directory. *If it contains the file "SYSTEM.STARTUP", you may proceed directly to topic "B".* If you do not have this file, here is how to create it:

Place Apple Pascal disks APPLE1 and APPLE2 in Drive 1 and Drive 2, respectively. (*Note: Disk APPLE1 must not be write-protected.*) Turn on the computer. From Command Level, choose to **E**(dit. Enter this program exactly as follows. (*Note: You may begin program lines in Column one in PASCAL.*):

```
PROGRAM INITFORTRAN;
USES APPLESTUFF;
BEGIN
END.
```

When you've finished typing it and have pressed CTL-C, press the *three* keys **Q U R** (*Note: PASCAL has no "Listing file ?" prompt to be answered; you will also notice that the Linker is not called in after compilation*). After compiling, enter the **F**(iler. Replace disk APPLE2 with disk FORT1. **T**(ransfer file **APPLE1:SYSTEM.WRK.CODE** to file **FORT1:SYSTEM.STARTUP.** (There is no need to save the Text version.) That's it!

This new program will be executed every time you boot FORTRAN (because of its special file name), and will enable the use of APPLESTUFF and TURTLEGRAPHICS. (*Note: The greeting message normally given when*

you boot will no longer be seen once file SYSTEM.STARTUP has been supplied.)

B. ACCESSING APPLESTUFF

To use any of APPLESTUFF's routines in a program, you must place this Compiler Directive at the start of your program:

```
$       USES APPLESTUFF
```

NOTE: APPLESTUFF is an overlay of SYSTEM.LIBRARY. In other words, it is not linked into your program, but rather is read in at execution time. Therefore, FORT1 must be present in order to run a program that incorporates APPLESTUFF.

C. THE "RANDOM" FUNCTION

RANDOM is a *function* which returns a randomly generated Integer between 0 and 32,767 (inclusive). Unlike the other functions we have studied, *"RANDOM" has no arguments (yet it requires a pair of dummy parentheses).* Here is a program segment which will return ten random values:

```
$       USES APPLESTUFF

        PROGRAM RNDM
        INTEGER COUNTR

        DO 2Ø COUNTR = 1, 1Ø
           WRITE (*, 1Ø) RANDOM ()
1Ø         FORMAT (I5)
2Ø      CONTINUE

        END
```

RANDOM has an inherent drawback: it returns the *same sequence* of random numbers every time it is called. The above program will keep printing the same ten numbers every time it is run! We will learn how to overcome this problem a bit later.

In addition to its use in WRITE, RANDOM may also be used in assignment statements and decisions, just as you would other functions:

```
NUMBER = RANDOM ()

IF (RANDOM () .GT. 1ØØØ) GOTO 5Ø
```

Let's address the problem of narrowing down the 32,768 possible random numbers to a specific range, say 1 through 10. To begin, we need to examine the MOD function. Recall that it requires two Integer arguments. The first number supplied to MOD is really a dividend, while the second value is a divisor. MOD returns the *remainder* of the division of these two numbers. For instance, MOD (25, 3) returns a 1, since 25 divided by 3 is 8 with a *remainder* of 1. In the same way, MOD (19, 4) evaluates to 3, and MOD (23, 6) returns a 5.

An interesting feature of taking say, MOD (any number, 5), is that *there are exactly as many possible remainders as the divisor you choose* (in this case, five possible remainders—0, 1, 2, 3, and 4), no matter how large the dividend is! In addition, *the remainders always begin at zero.*

Now let's say that out of the 32,768 possible values the RANDOM function returns, I would like only ten possible answers. How about taking MOD (RANDOM ( ), 10)? Isn't it true that this can return only 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 (the possible remainders when dividing by 10)? Sure it is! What if I would like the ten possible values to be in the range 1 through 10, as opposed to 0 through 9? If you said "add one to the value returned by MOD," give yourself a silver dollar!

This discussion leads us to the following formula for producing random numbers in a specific range:

```
lowest possible value + MOD (RANDOM (), number of possible values)
```

MOD serves to limit the number of possible answers returned, and, since these answers will always begin at 0, the addition shifts this range of values to a specific starting point (the lowest possible value).

Let's generate numbers from 1 through 6 for a dice simulation. We wish to start at 1, and there are a total of six possible random values we want returned, so we use:

```
ROLL = 1 + MOD (RANDOM (), 6)
```

How about random values from 12 through 15? That's four elements (count them again if you said 3), starting at 12, so we have:

```
VALUE = 12 + MOD (RANDOM (), 4)
```

D. THE "RANDOI" SUBROUTINE

RANDOI (short for RANDOmIze, although I would've named it RNDMZE) is a *subroutine* which resets RANDOM's reference point for generating a sequence of random numbers. RANDOI has no arguments (*but*

subroutines, unlike functions, do NOT require the use of dummy parentheses when there are no arguments). As is true of other subroutines, RANDOI must be preceded by the keyword "CALL".

As we just discussed, RANDOM always generates the *same* sequence of numbers. RANDOI changes RANDOM's initial reference point for generating the numbers, thereby causing a *different* sequence every execution (a random sequence of random numbers, if I may pun). We now modify the previous program using RANDOI:

```
$       USES APPLESTUFF

        PROGRAM RNDM
        INTEGER COUNTR
        CALL RANDOI

        DO 2Ø COUNTR = 1, 1Ø
           WRITE (*, 1Ø) RANDOM ()
1Ø         FORMAT (I5)
2Ø      CONTINUE

        END
```

Note that RANDOI is called only *once,* to scramble RANDOM's initialization. Once this has been done, RANDOM can handle the random number generation on its own.

E. THE "NOTE" SUBROUTINE

The NOTE subroutine has this form:

```
CALL NOTE (pitch, duration)
```

"Pitch" is an Integer between 0 and 50. A pitch of zero corresponds to a rest, while pitches 1–50 yield a tempered chromatic scale. The following conversion table was produced with the help of a tone generator:

Ø	Rest	7	B	14	F#,Gb
1	E#,F	8	B#,C	15	G
2	F#,Gb	9	C#,Db	16	G#,Ab
3	G	1Ø	D	17	A
4	G#,Ab	11	D#,Eb	18	A#,Bb
5	A	12	E	19	B
6	A#,Bb	13	E#,F	2Ø	B#,middle C

21	C#,Db	31	B	41	A
22	D	32	B#,C	42	A#,Bb
23	D#,Eb	33	C#,Db	43	B
24	E	34	D	44	B#,C
25	E#,F	35	D#,Eb	45	C#,Db
26	F#,Gb	36	E	46	D
27	G	37	E#,F	47	D#,Eb
28	G#,Ab	38	F#,Gb	48	E
29	A	39	G	49	E#,F
3Ø	A#,Bb	4Ø	G#,Ab	5Ø	F#,Gb

The above notes are *not* pure instrumental tones, and in general are about 1/4 flat.

"Duration" is an Integer between 1 and 255. The duration length is somewhat arbitrary, with "255" yielding a note held for roughly one second. The other Integers would of course yield shorter notes.

Let's try out Apple FORTRAN's scale:

```
$       USES APPLESTUFF

        PROGRAM SCALE
        INTEGER TONE

        DO 1Ø TONE = 1, 5Ø
           CALL NOTE (TONE, 1Ø)
1Ø      CONTINUE

        END
```

This loop will play each note of the scale, from lowest note (1) to highest (50), with duration 10.

One can produce some interesting sound effects with the NOTE subroutine. The following is my personal favorite:

```
$       USES APPLESTUFF

        PROGRAM ALIEN
C       Produces out-of-this-world sound!

        INTEGER TONE, SCALES

C       Play 5 rapid upward scales
        DO 2Ø SCALES = 1, 5
           DO 1Ø TONE = 1, 5Ø
              CALL NOTE (TONE, 1)
```

```
1Ø          CONTINUE
2Ø       CONTINUE

C        Play 3 rapid downward scales
         DO 4Ø SCALES = 1, 3
            DO 3Ø TONE = 5Ø, 1, -1
               CALL NOTE (TONE, 1)
3Ø          CONTINUE
4Ø       CONTINUE

         END
```

F. REVIEW QUESTIONS

1. A person tries to execute a program that "$USES APPLESTUFF". The screen becomes filled with garbage and the system hangs when s/he does so. What probably caused the problem?

2. Write a statement to generate a random value between 75 and 100 (inclusive), and assign it to a variable named GUESS.

3. What possible values may variable NUMBER be assigned in the following statement?

```
NUMBER = -5 + MOD (RANDOM (), 11)
```

4. Do you think it would be possible for a person to write a program using the NOTE subroutine to play harmonies, or even chords?

G. EXERCISES

1. Test the "randomness" of the random number generator by generating 1000 random values from 0 through 9. Have the program count and report the number of occurrences of each value. Perhaps you would like to run several trials and do statistical analyses (like mean and standard deviation) on the results.

2. Random numbers open up loads of possible simulations! Here's one: a program that simulates a cross-country runner's time over a specified course. The program takes into account the terrain of the course and the condition

of the runner. The times for a given interval are generated randomly in this manner (times are in minutes and seconds):

TERRAIN	DISTANCE	POOR SHAPE	GOOD SHAPE	EXCELLENT SHAPE
level	Ø.5Ø mi	3:15-4:ØØ	2:45-3:3Ø	2:15-3:ØØ
uphill	Ø.25 mi	2:3Ø-3:ØØ	2:15-2:45	2:ØØ-2:3Ø
level	1.ØØ mi	7:ØØ-8:ØØ	6:15-7:15	5:3Ø-6:3Ø
downhill	Ø.25 mi	1:45-2:25	1:3Ø-2:Ø5	1:15-1:45
level	1.ØØ mi	7:15-8:45	6:3Ø-7:45	5:45-6:45

Ask the program user to enter her/his condition, then begin the race! Describe the course intervals and report the time for each. When finished, report total course time.

Note that Integer Division coupled with Mod Division can convert a figure in *seconds* into *minutes and seconds* rather nicely. For example, let's say your program found that a runner ran the course in 1,315 seconds. You now want to find out how many 60-second intervals you can take out of 1,315 without regard for any remainder (*i.e., you want a quotient with no decimal component*). That's where Integer Division comes in. 1315 / 60 = 21 minutes. Now we must find the leftover seconds (*i.e., the remainder of the division*). That's where Mod Division comes in. MOD (1315, 60) = 55 seconds. So 1,315 seconds evaluates to 21 minutes and 55 seconds with two simple operations!

For a challenge, you may want to enter an entire team (or teams) and keep track of times for all runners. You may also want to add more intervals, or even set up meets for several different courses. (If you're really into it, take the windspeed and its direction into account.)

3. Write a program to accept a series of notes and play them back for you. A suggested approach would be to ask for the given note, pitch, and duration as follows:

```
NOTE   A, B, C, D, E, F, G, R(est or Q(uit

PITCH : F(lat N(atural S(harp

DURATION : S(ixteenth E(ighth Q(uarter H(alf W(hole
```

You will have to convert the first two parameters to the corresponding PITCH argument (0–50) for the NOTE subroutine. The DURATION argument (1–255) must also be determined. Save the pitch and duration values for each note in a Data file, then play them back after the entire tune has been entered.

You may want to add an octave parameter, or offer dotted notes. (Slurs are probably out of the question, however.)

Chapter 19:
THE TURTLEGRAPHICS LIBRARY UNIT

A. INTRODUCTION

Apple FORTRAN's System Library contains a graphics package (developed by Seymour Papert at the Massachusetts Institute of Technology) called *TURTLEGRAPHICS*. Although originally invented to help elementary-school children learn how to use computer graphics, this package has been considerably expanded and is well worth studying. TURTLEGRAPHICS has been proposed as a possible standard for graphics. It is already incorporated in the computer language LOGO.

TURTLEGRAPHICS offers several advantages over standard Applesoft Hi-res graphics. Here are just a few:

1. It is not coordinate-oriented. This makes it easier to produce figures, especially for non-mathematically-oriented people.

2. Those who prefer coordinate graphics need not feel alienated, for drawings may also be made by specifying coordinates. These people will be pleased to find that the origin is at the *lower left* of the screen, similar to conventional mathematical coordinate systems. X coordinates range from 0 to 279 on the TURTLEGRAPHICS screen, while Y coordinates range from 0 to 191.

3. You may put text anywhere on the graphics screen (*not just on the four bottom lines*) without having to learn shape tables.

4. You may alter the graphics "window" to any size you desire with a straightforward, easily understood command (as opposed to a series of obscure POKE incantations).

B. ACCESSING TURTLEGRAPHICS

TURTLEGRAPHICS is referenced with the following Compiler Directive:

```
$       USES TURTLEGRAPHICS
```

TURTLEGRAPHICS is an overlay of the System Library, so it is *not* linked with your program. Since it *is* read in at execution time, FORT1 must be present to access TURTLEGRAPHICS.

C. BACKGROUND

TURTLEGRAPHICS is based on the wanderings of an imaginary turtle. This turtle may be turned in any direction and then moved *(directly forward or backward only).* As it moves, it drags behind it an imaginary pen, thus producing lines. *It is possible to produce an entire drawing without ever specifying a single coordinate.*

D. COMMANDS

Unless otherwise specified, all commands are *subroutines,* and as such must be prefaced with the keyword CALL. The six-letter subroutine name limit results in some rather strange-looking abbreviations. The text will explain their derivation.

1. INITTU

INITTU (INITialize TUrtlegraphics) calls in and *clears* the graphics screen. It then places the turtle in the center of the screen (coordinates: 139, 95) facing right (angle: 0 degrees).

In addition, the pen color is set to black, and the graphics window is set to full screen.

INITTU has no parameters, so it looks like this:

```
CALL INITTU
```

2. GRAFMO

GRAFMO (GRAFics (sic) MOde) also switches from text to graphics, *but does no initialization.* In other words, the graphics screen is left *intact.* This is useful when switching back and forth from text to graphics repeatedly in the same program.

GRAFMO has no parameters, and looks like this:

```
CALL GRAFMO
```

3. TEXTMO

TEXTMO (TEXT MOde) switches from the graphics screen to the text screen, and has no parameters. The nice thing is that although TEXTMO leaves graphics mode, *it does not erase the graphics screen in memory.* If you later use GRAFMO, the graphics screen will be brought up exactly as it appeared when you exited with TEXTMO. Here's how this command looks:

```
CALL TEXTMO
```

4. VIEWPO

VIEWPO (VIEWPOrt) sets the dimensions of the graphics "window." Here is the form of this command:

```
CALL VIEWPO (left, right, bottom, top)
```

"left" is the leftmost X coordinate to display
"right" is the rightmost X coordinate to display
"bottom" is the bottommost Y coordinate to display
"top" is the topmost Y coordinate to display

NOTE: YOU WILL NOT GENERATE A RUN-TIME ERROR IF YOU ATTEMPT TO DRAW OUTSIDE THE GRAPHICS WINDOW. This is true even if the graphics window is set to full screen (the default "VIEWPO"). Therefore, it is recommended that you use error-trapping in your program to prevent off-screen plotting from occurring. Otherwise it is very easy to "lose" the turtle somewhere beyond the viewport!

5. PENCOL

PENCOL (PENCOLor) sets the ink color for the turtle's pen. Its form follows:

```
CALL PENCOL (color)
```

The available colors in TURTLEGRAPHICS are:

```
Ø  invisible ink        5  black1        9  black2
1  white                6  green        1Ø  orange
2  black                7  violet       11  blue
3  reverse              8  white1       12  white2

4  unassigned
```

Some colors require a brief explanation. Recall that in TURTLEGRAPHICS, lines are drawn as an imaginary turtle moves with a pen "in tow." *Invisible ink* allows you to move over *any* color background without leaving a line (you may think of it as lifting the pen). *Reverse* is just the opposite: it allows you to move over *any* color and leave a line; tracing over black leaves a white line, tracing over white leaves a black line, etc. *These two colors eliminate the need to change pen color every time you meet a different background color while drawing a line.*

You may wonder why you have three versions of white and black from which to choose. It turns out that a line drawn in any of the colors (green, violet, orange, or blue) requires a greater width (i.e., a greater number of pixels) than a normal white or black line, in order that it may appear with a high degree of contrast. *Therefore, one may not fully erase a green or violet line by tracing over it in black.* That's where *Black1* comes in; it's a *wider* black than normal! Likewise, White2 and Black2 should be used to overstrike an orange or blue line.

To make color selection less confusing, the colors are arranged in groups of four (colors 0–3, colors 5–8, and colors 9–12). In each group of four colors, the first and last colors are the "special colors" for use in that set, like Black1 and White1. The two inner colors happen to be color negatives, like green and violet.

6. FILLSC

FILLSC (FILL SCreen) fills the *entire viewport* with the given color (using the same color codes given above):

```
CALL FILLSC (color)
```

Two useful commands: "FILLSC (2)" clears the entire viewport; "FILLSC (3)" gives you a "negative" of your viewport. FILLSC is also very handy for setting background colors in a drawing.

7. MOVE

This will move the turtle forward (if you specify a positive length) or backward (negative length) along the line in the direction it is facing:

```
CALL MOVE (length)
```

"Length" is an *Integer* specifying how far to move.

NOTE: "MOVE" is relative to the present turtle position. It is therefore important to know where the turtle is at all times. Remember: It is possible to move off the screen, so be careful!

8. MOVETO

Mathematicians were teethed on coordinate graphics, so this command is especially for them (lest they become disoriented):

```
CALL MOVETO (X coordinate, Y coordinate)
```

"CALL MOVETO (20, 40)" will therefore move the turtle to (20, 40) *no matter what direction the turtle is facing, and no matter how far it is from the present turtle position.* With MOVETO, it is possible to write a program entirely with Cartesian coordinates.

NOTE: Like MOVE, MOVETO moves from the present turtle position.

9. TURN

This subroutine allows you to change the direction the turtle is facing:

```
CALL TURN (angle)
```

"angle" is the number of degrees to rotate the turtle from the direction it is currently facing *(positive angle = counterclockwise, negative angle = clockwise).*

10. TURNTO

Whereas TURN is a *relative* command (it depends on what angle the turtle is currently facing), TURNTO is an *absolute* command (turn to the specified angle no matter *what* direction your turtle is now facing). Here is the form:

```
CALL TURNTO (angle)
```

To clear up the differences between the two apparently identical commands, let's say your turtle is facing left (180 degrees). Here are two ways to move it so that it is facing up (90 degrees):

```
CALL TURN (-90)   turn clockwise 90 degrees

CALL TURNTO (90)  turn so you are at a 90 degree angle
```

One more example; the turtle's current angle is 43, and you would like it to become 75:

```
CALL TURN (32)    turn counterclockwise 32 degrees
```

```
CALL TURNTO (75)
```
make current angle 75 degrees

NOTE: Neither of the two "TURN" commands affects the graphics screen. They merely change the direction the turtle faces.

11. Finding the turtle's present orientation

By now you should realize how crucial it is to know the turtle's position and direction. There are three *functions* which help you get your bearings *(although none of them have any arguments, they require a pair of dummy parentheses, just as any other function):*

```
TURTLX ()
```

(TURTLe's X coordinate) no parameters, returns the turtle's X coordinate

```
TURTLY ()
```

(TURTLe's Y coordinate) no parameters, returns the turtle's Y coordinate

```
TURTLA ()
```

(TURTLe's Angle) no parameters, returns the angle the turtle is facing

Again, these are *functions,* so they are *not* addressed with "CALL". Here is a valid usage of these three functions:

```
      WRITE (*, 1Ø) 'X: ', TURTLX (), 'Y: ', TURTLY (),
     *              'Angle: ', TURTLA ()
1Ø    FORMAT (A, I3, 5X, A, I3, 5X, A, I3)
```

12. WCHAR

WCHAR (Write CHARacter on graphics screen) allows you to place *one* character on the graphics screen with its lower left corner at the current turtle position. As the character is placed on the screen, *WCHAR moves the turtle seven units over on the X-axis* (since each character is seven units wide by eight units high). Here is the form:

```
CALL WCHAR ('character')
```

"character" is any *single* keyboard character enclosed in single quotes.

NOTES:

1. Unfortunately, the argument may not be a variable, nor even a word (i.e., you may call only one letter at a time).

2. It is not necessary to worry about the pen color when using "WCHAR". The character is automatically printed as a white character on a black background.

13. Keeping the graphics screen "up"

If you end a TURTLEGRAPHICS program with the word "END", the program will disappear almost the instant it appears. That is because "END" has an automatic "TEXTMO" built into it. To prevent the graphics screen from disappearing before you can examine it, use this little trick: *ask the program user to press "RETURN" to go back to text.* This gives him all the time he or she wants to look over his or her drawing. The FORTRAN code to handle this:

```
      READ (*, 20) DELAY
20    FORMAT (A)
```

Recall that "READ" waits until the "RETURN" key is pressed before allowing the program to proceed. This serves as our makeshift delay! Even though "READ" is expecting a value for the variable "DELAY", the program user may press "RETURN" without typing a value. The "A" format specifier is used in case the user inadvertently presses a key *before* pressing "RETURN".

E. TWO SAMPLE PROGRAMS

Let's kick in TURTLEGRAPHICS, outline the screen with a white line, then place the word "hello" in the middle of the screen. We'll finish by making a few screen negatives for a fancy touch:

PROGRAM LINES	COMMENTS
`$  USES TURTLEGRAPHICS`	Make sure FORT1 is on-line.
`PROGRAM DEMO`	
`CHARACTER DELAY*1`	

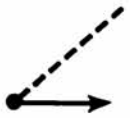
```
      CALL INITTU                    clear screen, turtle mid-screen
      CALL MOVETO (Ø, Ø)             move turtle to lower left
      CALL PENCOL (1)                why not the line before?



      CALL MOVE (279)                bottom border
      CALL TURN (9Ø)                 now facing up
      CALL MOVE (191)                right border
      CALL TURN (9Ø)                 facing left (180 deg.)
      CALL MOVE (279)                top border
      CALL TURN (9Ø)                 facing down (270 deg.)
      CALL MOVE (191)                left border



      CALL PENCOL (Ø)                set color to black before moving
      CALL MOVETO (123, 91)          move toward middle of screen
      CALL WCHAR ('h')               begin writing letters
      CALL WCHAR ('e')
      CALL WCHAR ('l')
      CALL WCHAR ('l')
      CALL WCHAR ('o')



      DO 1Ø NEGTVE = 1, 4            make 4 screen negatives
         CALL FILLSC (3)
1Ø    CONTINUE



      READ (*, 2Ø) DELAY             delay until <ret> is pressed
2Ø    FORMAT (A)

      END
```

NOTE: It would be illegal to use the line "CALL WCHAR ('hello')" in the above program. Do you know why?

a. CALL INITTU

Turtle at mid-screen, facing right; graphics screen empty.

b. CALL MOVETO (0,0)

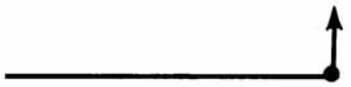


Turtle moved to origin, but left no trace, since default pencolor is 0 (invisible ink). It still faces right (zero degrees).

c. CALL PENCOL (1)
CALL MOVE (279)

Turtle moves 279 units in set direction with a white pen.

d. CALL TURN (90)

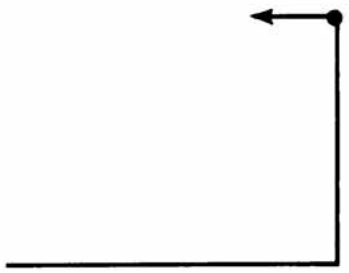


Turtle turns 90° counterclockwise.

e. CALL MOVE (191)

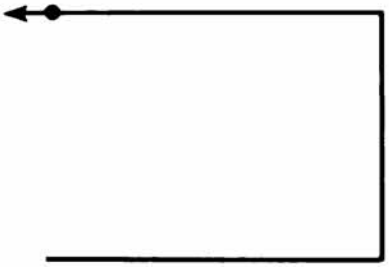
Turtle moves 191 units in set direction.

f. CALL TURN (90)



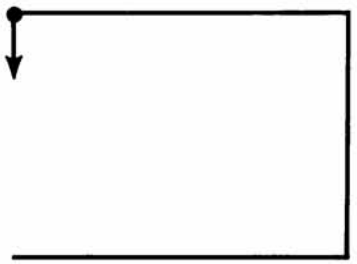
Turtle turns 90° and now faces left (180°).

g. CALL MOVE (279)



Turtle moves 279 units in set direction.

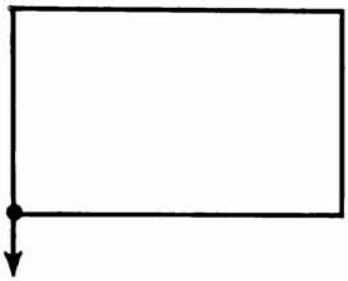
h. CALL TURN (90)



Turtle rotates 90° and now faces down (270°).

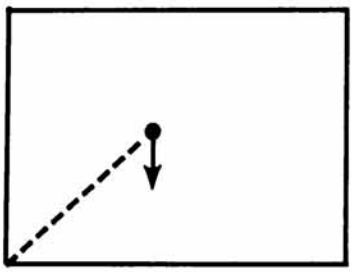
Figure 19.1 Turtle Travels
(continued)

i. `CALL MOVE (191)`



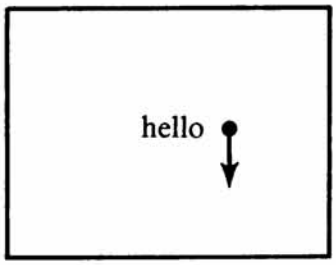
Turtle moves 191 units to complete the frame.

j.
```
CALL PENCOL (Ø)
CALL MOVETO (123,91)
```



Turtle moves invisibly to (123, 91).

k.
```
CALL WCHAR ('h')
CALL WCHAR ('e')
CALL WCHAR ('l')
CALL WCHAR ('l')
CALL WCHAR ('Ø')
```



Each character is drawn with its lower left corner on the turtle. The turtle moves over 7 units for each character drawn.

l.
```
    DO 1Ø NEGTVE = 1, 4
        CALL FILLSC (3)
1Ø  CONTINUE
```

I can't show you the four negatives on paper. Enter the program & see them for yourself!

Figure 19.1 Turtle Travels

Let's try another to draw a solid circle with its center at the middle of the screen, and a user-supplied radius. Note the simplicity of the algorithm, which merely changes the turtle's angle through each of the 360 degrees. At each angle, the turtle is moved out along a line of length RADIUS and then back again (to the center of the circle).

```
$       USES TURTLEGRAPHICS

        PROGRAM CIRCLE
        INTEGER RADIUS, ANGLE
        CHARACTER DELAY*1

5       WRITE (*, 1Ø)
     *        'Enter radius ( <= 95, format I2, or Ø to quit) : '
```

```
1Ø      FORMAT (/, A, $)
        READ (*, 15) RADIUS
15      FORMAT (I2)

        IF (RADIUS .GE. 1  .AND.  RADIUS .LE. 95) THEN
C          Clear graphics screen
           CALL INITTU
           CALL PENCOL (1)

C          Draw circle
           DO 2Ø ANGLE = 1, 36Ø
              CALL TURNTO (ANGLE)
              CALL MOVE (RADIUS)
              CALL MOVE (-RADIUS)
2Ø         CONTINUE

C          Delay until RETURN is pressed
           READ (*, 3Ø) DELAY
3Ø         FORMAT (A)
           CALL TEXTMO
        ENDIF

        IF (RADIUS .NE. Ø) GOTO 5
C       Otherwise
        END
```

F. REVIEW QUESTIONS

1. Why are there *two* TURTLEGRAPHICS commands to call in the graphics screen?

2. How many TURTLEGRAPHICS commands would be needed to draw a solid orange rectangle?

3. Why couldn't you predict how the TURTLEGRAPHICS screen would look after the command "CALL MOVE (10)" *without any further information?*

4. Why couldn't you predict how the TURTLEGRAPHICS screen would look after the command "CALL MOVETO (0, 0)" *without any further information?*

G. EXERCISES

1. To help yourself master TURTLEGRAPHICS, enter and execute the following interactive program. It gives you a chance to use every sub-

routine available in TURTLEGRAPHICS, and get immediate feedback on the effects of those commands.

```
$       USES TURTLEGRAPHICS

        PROGRAM GRAFIX
C       Turtlegraphics demonstration program

C       Specification statements
        INTEGER RINGS, CHOICE, COLOR, RIGHT, BOTTOM, TOP
        CHARACTER HOME*1, INVRSE*1, NORMAL*1, BELL*1
        DATA LEFT, RIGHT, BOTTOM, TOP, COLOR / Ø, 279, Ø, 191, Ø /

C       Special character definitions
        HOME = CHAR (12)
        INVRSE = CHAR (15)
        NORMAL = CHAR (14)
        BELL = CHAR (7)
        ASSIGN 2Ø TO MENU

C       Title page
        WRITE (*, 5) HOME
5       FORMAT (A, $)
        WRITE (*, 1Ø) 'Welcome to'
1Ø      FORMAT (/, /, /, 22X, A, /)
        WRITE (*, 15) INVRSE,
     *    '*** LEARNING APPLE FORTRAN TURTLEGRAPHICS ***', NORMAL
15      FORMAT (5X, A, A, A)

C       Menu
2Ø      CALL HITKEY
        CALL TEXTMO
        WRITE (*, 5) HOME
        WRITE (*, 25)
     *    'Asterisked menu options incorporate an automatic ''Grafmo''.'
25      FORMAT (A, /, /)
        WRITE (*, 25)
     *    'To return to the menu after any option, press RETURN.'

C       Available options
        WRITE (*, 3Ø) '* 1. Inittu', '  2. Grafmo', '  3. Viewpo'
3Ø      FORMAT (1ØX, A, /, 1ØX, A, /, 1ØX, A)
        WRITE (*, 3Ø) '  4. Pencol', '* 5. Fillsc', '  6. Turn'
        WRITE (*, 3Ø) '  7. Turnto', '* 8. Move', '* 9. Moveto'
        WRITE (*, 3Ø) '*1Ø. Wchar', ' 11. Report position', ' 12. End'
```

```
      WRITE (*, 4Ø) 'Your choice (1-12, format I2) ? '
4Ø    FORMAT (/, /, A, $)
      READ (*, 5Ø) CHOICE
5Ø    FORMAT (I2)

      WRITE (*, 5) HOME
      GOTO (6Ø, 7Ø, 8Ø, 9Ø, 1ØØ, 11Ø) CHOICE
      GOTO (12Ø, 13Ø, 14Ø, 15Ø, 16Ø, 17Ø) CHOICE - 6

C     Otherwise invalid choice
      DO 55 RINGS = 1, 5
         WRITE (*, 5) BELL
55    CONTINUE

      WRITE (*, 5) 'Invalid reply...'
      GOTO MENU

C     Initialize graphics screen
6Ø    CALL INITTU
      GOTO MENU

C     Display graphics screen
7Ø    CALL GRAFMO
      GOTO MENU

8Ø    CALL VWPORT (LEFT, RIGHT, BOTTOM, TOP)
      GOTO MENU

9Ø    CALL PENCLR (CHOICE, COLOR)
      GOTO MENU

1ØØ   CALL PENCLR (CHOICE, COLOR)
      GOTO MENU

11Ø   CALL RELROT
      GOTO MENU

12Ø   CALL ABSROT
      GOTO MENU

13Ø   CALL RELMOV
      GOTO MENU

14Ø   CALL ABSMOV
      GOTO MENU
```

```
15Ø     CALL TEXT
        GOTO MENU

16Ø     CALL REPORT (LEFT, RIGHT, BOTTOM, TOP, COLOR)
        GOTO MENU

17Ø     END

        SUBROUTINE HITKEY
C       Delays until program user presses RETURN

        CHARACTER DELAY*1

        WRITE (*, 1Ø) 'Press RETURN to continue...'
1Ø      FORMAT (/, /, /, 14X, A, $)
        READ (*, 2Ø) DELAY
2Ø      FORMAT (A)

        END

        SUBROUTINE VWPORT (LEFT, RIGHT, BOTTOM, TOP)
C       Set viewport

        INTEGER RIGHT, BOTTOM, TOP

5       WRITE (*, 1Ø) 'Leftmost graphable X-coordinate (I3) ? '
1Ø      FORMAT (/, A, $)
        READ (*, 2Ø) LEFT
2Ø      FORMAT (I3)

        IF (LEFT, .LT. Ø  .OR.  LEFT .GT. 279) THEN
           WRITE (*, 1Ø) 'Must be between Ø and 279...'
           GOTO 5
        ENDIF

3Ø      WRITE (*, 1Ø) 'Rightmost graphable X-coordinate (I3) ? '
        READ (*, 2Ø) RIGHT

        IF (RIGHT, .LT. LEFT) THEN
           WRITE (*, 1Ø) 'Rightmost must be >= leftmost...'
           GOTO 3Ø
        ELSEIF (RIGHT .GT. 279) THEN
           WRITE (*, 1Ø) 'Must be <= 279...'
           GOTO 3Ø
        ENDIF
```

```
4Ø      WRITE (*, 1Ø) 'Bottommost graphable Y-coordinate (I3) ? '
        READ (*, 2Ø) BOTTOM

        IF (BOTTOM .LT. Ø  .OR.  BOTTOM .GT. 191) THEN
          WRITE (*, 1Ø) 'Must be between Ø and 191...'
          GOTO 4Ø
        ENDIF

5Ø      WRITE (*, 1Ø) 'Topmost graphable Y-coordinate (I3) ? '
        READ (*, 2Ø) TOP

        IF (TOP .LT. BOTTOM) THEN
          WRITE (*, 1Ø) 'Topmost must be >= bottommost...'
          GOTO 5Ø
        ELSEIF (TOP .GT. 191) THEN
          WRITE (*, 1Ø) "Must be <= 191..'
          GOTO 5Ø
        ENDIF

        CALL VIEWPO (LEFT, RIGHT, BOTTOM, TOP)

        END

        SUBROUTINE PENCLR (CHOICE, COLOR)
C       Set pen color

        INTEGER CHOICE, COLOR

C       Report color codes
        WRITE (*, 1Ø) 'The color codes are:'
1Ø      FORMAT (A, /)
        WRITE (*, 2Ø) 'Ø. Invisible ink       7. Violet'
2Ø      FORMAT (A)
        WRITE (*, 2Ø) '1. White               8. White 1'
        WRITE (*, 2Ø) '2. Black               9. Black 2'
        WRITE (*, 2Ø) '3. Reverse            10. Orange'
        WRITE (*, 2Ø) '4. Unused             11. Blue'
        WRITE (*, 2Ø) '5. Black 1            12. White 2'
        WRITE (*, 2Ø) '6. Green

3Ø      WRITE (*, 4Ø) 'Pen color desired (Ø - 12, format I2) ? '
4Ø      FORMAT (/, /, A, $)
        READ (*, 5Ø) COLOR
5Ø      FORMAT (I2)
```

```
      IF (COLOR .LT. Ø  .OR.  COLOR .GT. 12) THEN
         WRITE (*, 1Ø) 'Invalid reply...'
         GOTO 3Ø
      ENDIF

      IF (CHOICE .EQ. 4) THEN
         CALL PENCOL (COLOR)
      ELSE
         CALL GRAFMO
         CALL FILLSC (COLOR)
      ENDIF

      END

      SUBROUTINE RELROT
C     Relative turtle rotation

      INTEGER ANGLE

5     WRITE (*, 1Ø) 'Turn how many degrees (I3) ? '
1Ø    FORMAT (/, A, $)
      READ (*, 2Ø) ANGLE
2Ø    FORMAT (I3)

      IF (ANGLE .LT. -36Ø  .OR.  ANGLE .GT. 36Ø) THEN
         WRITE (*, 1Ø) 'Must be between -36Ø and +36Ø...'
         GOTO 5
      ENDIF

      CALL TURN (ANGLE)

      END

      SUBROUTINE ABSROT
C     Absolute turtle rotation

      INTEGER ANGLE

5     WRITE (*, 1Ø) 'Turn to which angle (I3) ? '
1Ø    FORMAT (/, A, $)
      READ (*, 2Ø) ANGLE
2Ø    FORMAT (I3)

      IF (ANGLE .LT. Ø  .OR.  ANGLE .GT. 36Ø) THEN
```

```
            WRITE (*, 1Ø) 'Must be between Ø and 36Ø...'
            GOTO 5
         ENDIF

         CALL TURNTO (ANGLE)

         END

         SUBROUTINE RELMOV
C        Relative turtle move

         INTEGER DSTNCE

         WRITE (*, 1Ø) 'Move how far (I3) ? '
1Ø       FORMAT (A, $)
         READ (*, 2Ø) DSTNCE
2Ø       FORMAT (I3)
         CALL GRAFMO
         CALL MOVE (DSTNCE)

         END

         SUBROUTINE ABSMOV
C        Absolute (coordinate) turtle move

         INTEGER X, Y

5        WRITE (*, 1Ø) 'X-coordinate to move to (I3) ? '
1Ø       FORMAT (/, A, $)
         READ (*, 2Ø) X

2Ø       FORMAT (I3)

         IF (X .LT. Ø  .OR.  X .GT. 279) THEN
            WRITE (*, 1Ø) 'Must be between Ø and 279...'
            GOTO 5
         ENDIF

3Ø       WRITE (*, 1Ø) 'Y-coordinate (I3) ? '
         READ (*, 2Ø) Y

         IF (Y .LT. Ø  .OR.  Y .GT. 191) THEN
            WRITE (*, 1Ø) 'Must be between Ø and 191...'
```

```
            GOTO 3Ø
         ENDIF

         CALL GRAFMO
         CALL MOVETO (X,Y)

         END

         SUBROUTINE TEXT
C        Place asterisk on screen

         CHARACTER RETURN*1

         WRITE (*, 1Ø) 'An asterisk will be placed on the'
1Ø       FORMAT (A)
         WRITE (*, 2Ø) 'graph when you press RETURN. . .'
2Ø       FORMAT (A, $)
         READ (*, 3Ø) RETURN
3Ø       FORMAT (A)
         CALL GRAFMO
         CALL WCHAR ('*')

         END

         SUBROUTINE REPORT (LEFT, RIGHT, BOTTOM, TOP, COLOR)
C        Report viewport, turtle position, and pen color

         INTEGER RIGHT, BOTTOM, TOP, COLOR

         WRITE (*, 1Ø) 'VIEWPORT'
1Ø       FORMAT (/, /, A, /)
         WRITE (*, 15) 'Left : ', LEFT
15       FORMAT (5X, A, I3)
         WRITE (*, 15) 'Right : ', RIGHT
         WRITE (*, 15) 'Bottom : ', BOTTOM
         WRITE (*, 15) 'Top : ', TOP

         WRITE (*, 1Ø) 'TURTLE POSITION'
         WRITE (*, 15) 'X-coordinate : ', TURTLX ()
         WRITE (*, 15) 'Y-coordinate : ', TURTLY ()
         WRITE (*, 15) 'Angle : ', TURTLA ()

         WRITE (*, 2Ø) 'PEN COLOR : ', COLOR
2Ø       FORMAT (/, /, A, I2)

         END
```

2. Modify the circle-producing program so that only the *outline* of the circle appears.

3. Write a "kaleidoscope" program by using the random number generator to select a random viewport. The FILLSC subroutine can then "paint" it with a randomly generated color. After ten such rectangles have been displayed, give the program user an option to continue (by generating ten more rectangles), or to quit.

4. Write a program that produces a pie graph representing the percentages of a company's expenditures in each of four areas: payroll, materials, research, and investments. The user prompts and replies should look something like this:

```
Enter total company expenditures for each of the four
areas listed below; the input format is F9.2

* Payroll $254100.00
* Materials $452000.00
* Research $678400.00
* Investments $127000.00

(clear the screen)
Your company expenditures totalled $1511500.00
The percentages for each of the four areas were

* Payroll      17%
* Materials    30%
* Research     45%
* Investments   8%

Press RETURN to see the color coded pie graph. . .
```

Note that the program calculated total company expenditures and percentages *before* showing the graph. The actual pie graph should have a color key. To create one, simply graph a small square of the color for say, Research, and place the abbreviation "Res" next to it. (Placing the entire word "Research" would be rather tedious, given the WCHAR subroutine's one-character-per-call limit.) Likewise, if the violet portion of the pie represents Investments, place a small violet square on the screen with the code "Inv" next to it. The graph should remain until the program user presses RETURN, after which he or she should be given a chance to graph a new set of data.

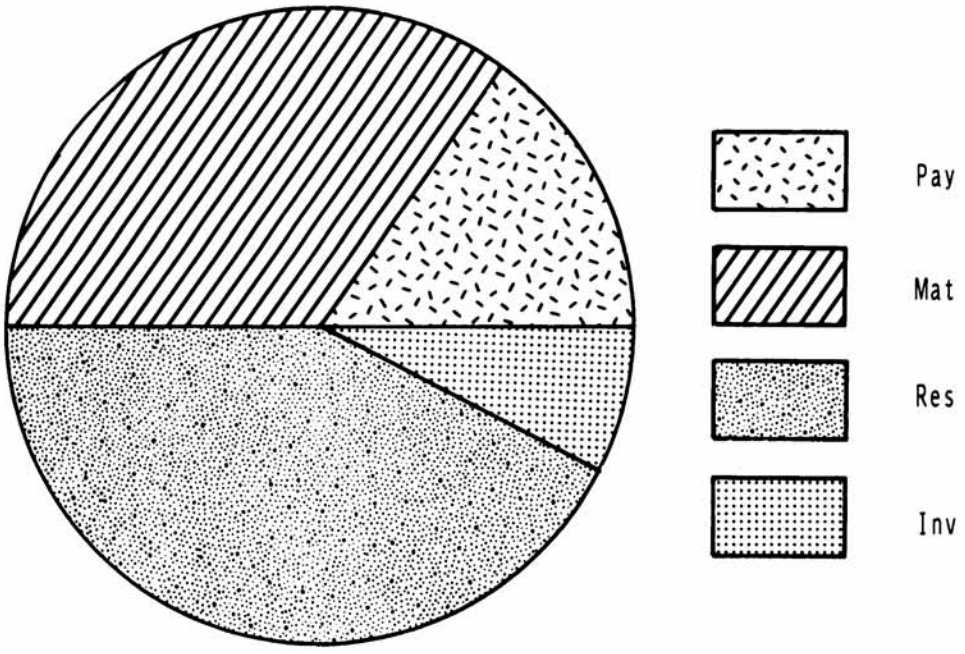


Figure 19.2 Pie Graph Program Graphics Output

A rather difficult extension would be to place the percentage figures (17%, 30%, etc.) right in the pie (as opposed to reporting them on the text screen before the graph is generated). If you attempt to code this esoteric feature, remember that WCHAR cannot print variables!

5. Modify the above program so that the four percentages are reported in bar graph form.

Chapter 20:
WORD PROCESSING WITH THE EDITOR

A. INTRODUCTION

Word processing involves using a computer as the *ultimate* typewriter. Computers have memories, and memories allow one to get a document perfect *before* printing it. How? That's a loaded question, and it will require the rest of this chapter to answer.

Believe it or not, you're already familiar with several word processing techniques, for the Apple FORTRAN Editor *is* a word processor. The problem is that we've used it only to write FORTRAN programs. Now we're going to use it to write letters, memos, term papers (!), etc.

B. SETTING UP THE EDITOR FOR WORD PROCESSING APPLICATIONS

Recall that the Editor is on disk FORT2. Call it in from Command Level and you'll see the familiar Editor prompt line. There happens to be a hidden option in that line. Type **"S"** and you'll see:

```
>SET: E(NVIRONMENT M(ARKER <ESC>
```

1. SET MARKER

A marker is an invisible sequence of characters you may place in any file created by the Editor. Simply move the cursor to the desired marker position, press **S**(et and **M**(arker, and you'll see:

```
SET WHAT MARKER?
```

You may now type up to *eight keyboard characters* to serve as your marker. The marker will be placed in the file *but you won't be able to see it.* Since markers are invisible, I strongly recommend using simple ones like A, B, and

C. That way they're easy to remember, and so is the order in which they were placed in the file.

How do you find a marker if you can't see it? Two ways, both of which you've seen before in the Editor prompt line:

a. JUMP: B(EGINNING E(ND M(ARKER <ESC>
Since you may JUMP to a marker, a good place to put one would be at the beginning of a paragraph which is giving you trouble, and which will probably require later re-writing. Surprisingly, JUMP will find your marker no matter what direction the prompt line arrow faces.

b. COPY: B(UFFER F(ROM FILE <ESC>
Not only can you copy from a file, you may copy only a *portion* of a file between markers. Let's say you're ambitious and are writing a novel. A good place for markers would be at the beginning of each new chapter. If you were wise enough to use simple markers, and wanted to copy only Chapter 6 of your masterpiece, you could use this response to the "COPY FROM FILE" prompt:

```
>COPY: FROM WHAT FILE [MARKER, MARKER]? ZELMO:NOVEL.TEXT [6,7]
```

In other words, copy all text between the marker "6" and the marker "7". *(Note the square brackets. Apple II Plus users will need to press CTL-K and SHIFT-M to generate these characters.)* You may use up to *ten markers* per file. What if your novel is fifteen chapters and you'd like markers preceding each? Simply split it into two files. (Aside: Actually, you'd get nowhere near ten chapters in a single file anyway, but you get the point.)

2. SET ENVIRONMENT

"ENVIRONMENT" refers to the word processing environment. Choose S(ET and E(NVIRONMENT and you'll see:

```
>ENVIRONMENT: [OPTIONS]  <ETX> OR <SP> TO LEAVE

      A(UTO INDENT          TRUE
      F(ILLING              FALSE
      L(EFT MARGIN          Ø
      R(IGHT MARGIN         78
      P(ARA MARGIN          5
      C(OMMAND CH           ^
      T(OKEN DEF            TRUE
```

```
4 BYTES USED, 17404 AVAILABLE

PATTERNS:
<TARG> = 'ENND'   <SUBST> = 'END'

MARKERS:
A
```

Note that you are given a list of alterable parameters, available memory information, patterns used in the most recent R(eplace, and a list of file markers. Let's take the parameters one by one.

a. *AUTO INDENT* refers to where the cursor positions itself when you press RETURN. When you are writing programs in FORTRAN, it's nice to have it return to the indentation level of the previous line. That's what happens when this value is TRUE. ENGLISH text does *not* always have to start in column 7, as you're probably aware. There's no reason for AUTO INDENT to be operative, so we'll shut it off. Type "**A**" (for AUTO INDENT) and "**F**" (for FALSE) to do so. *The cursor will now always return to the set left margin.*

b. *FILLING* is set to false. This prevents line "wrap around" (a real no-no in strict, 72-column FORTRAN). Typing "**FT**" will set FILLING to TRUE. Now you're set for a little magic. After exiting the Environment, *you may type as long as you wish and never press RETURN. Not only will the cursor wrap around, your words will never be chopped off at the end of the screen.* FILLING means "fill as many words as possible between the set margins." If you're on the 4th letter of a 7-letter word when you hit the right margin, *the entire word is re-written automatically on the next line.* This is truly a touch-typist's dream!

c. *LEFT MARGIN* and *RIGHT MARGIN* are self-explanatory. When you do change them, FILLING will automatically incorporate the new margins for you as you type.

d. *PARAGRAPH MARGIN* is the indentation automatically given the first line of a new paragraph *(which is defined as any text with a blank line preceding it and following it).* The *only* time you should press RETURN when word processing is at the end of your paragraph. (Press it *twice* to generate the blank line.) It is perfectly legal to set your paragraph margin *less than* your left margin, thereby "outdenting" your topic sentence. Again, once the paragraph margin is set, FILLING will automatically enforce it for you as you type.

e. *COMMAND CHARACTER* serves as a "keep your hands off this line" message. If you put "∧" as the first character of any line, that line will be *protected from being margined.* This is invaluable for making sure carefully formatted text (like a table) doesn't get re-arranged accidentally into paragraph form.

f. *TOKEN DEFAULT* can be set to FALSE if you want all FINDs and REPLACESs to use *Literal* (exact character sequence) search. (Recall that *Token* is normally the default.)

g. To exit the Environment, press either CTL-C or the space bar.

C. OTHER WORD PROCESSING COMMANDS

1. M(ARGIN

This is another Editor option that doesn't appear in the prompt line. Let's say you've just finished typing a paragraph, but don't like its format. Do you have to type it over several times with different margins until you're happy? Nope. Go back to the Environment, change the margins, leave the Environment, and press "**M**". *(NOTE: "MARGIN" will work only when "AUTO INDENT" is false and "FILLING" is true.)* The screen will go blank for a second or two, then reappear. The current paragraph will now be displayed with the new margins. Unfortunately, *MARGIN will never affect more than one paragraph.* If you wish to re-margin several paragraphs, you must do each individually. (Move the cursor anywhere in the paragraph, press "**M**", move the cursor to a new paragraph, press "**M**", etc.)

2. COPY

The C(OPY B(UFFER option is one of the most useful word processing options. If you want to move a sentence or paragraph around, delete it (to get it into the buffer), move the cursor to a new position, and press "**CB**".

3. REPLACE

This option allows you to develop your own shorthand. This has many applications, but one of the most common is the form letter. If you want to send one to fifteen people, you could type it and use "***" in place of a name. Your letter might begin something like this:

```
Dear ***,
   We are pleased to announce the graduation of our son, Zelmo.
Since you're such a close family-friend ***, we'd like to ask
you to join our celebration, etc.
```

To send a copy to your uncle Tudbury, you simply load the file, type "**/R**" (for infinite Replace), then "**/***//Tudbury/**" (thereby replacing all occurrences of "***" with "Tudbury").

4. ADJUST

This will adjust the current line to either the left margin, the right margin, or, most usefully of all, to the center of the page (midway between the two set margins). To center a line, move the cursor anywhere in the line and press "**AC**" (Adjust Center).

D. WARNING

Remember that when you're editing a file, everything is contained in memory. Save your file frequently to avoid disaster. When at last you're satisfied, save the final version and use the Filer to transfer your perfect(!) document to the Printer. You will be happy to find out that the Editor is a *what-you-see-is-what-you-get* word processor, in that the screen is dumped unaltered into the file, and subsequently, to the Printer.

E. DISADVANTAGES

Although the Apple FORTRAN Editor offers many useful word processing features, it lacks a few others:

1. There is no automatic paging. Many word processors will automatically leave a few blank lines and advance to the next page during printout of longer files (most even number the pages for you). If your document is more than one page in length, you must supply page breaks of your own by inserting blank lines. Luckily, many printers offer such a page-break feature (check your printer manual).

2. There is no Fill Justify option. This results in even left *and* right margins (as they are in a book or a magazine). The right margin will always be uneven when you use the Apple FORTRAN Editor.

3. There is no provision for double spacing of lines. If you try to insert blanks between lines, each line will be considered a new paragraph (why?). One way around this is to set your printer switches so that the printer generates a line feed at the end of each line. Normally this is not necessary, since the operating system sends its *own* line feed. However, these two line feeds (operating system + printer) result in double spacing.

F. DISADVANTAGES FOR APPLE II PLUS USERS

1. The Editor display is set up for eighty columns, so CTL-A or CTL-Z must be used to flip between left and right screens. A slick way around this is to set the right margin to 39, thereby insuring that you're always on the left half of the screen. Then, should you desire 80-column printout, set your right margin back to 78, and M(ARGIN your text.

2. The Editor will accept mixed-case text, but older Apples can generate only upper-case letters. The only solution to this problem (without using the CTL-E *simulated* case change feature) is the purchase of a relatively inexpensive lower-case chip.

G. REVIEW QUESTIONS

1. A student sets a marker at the beginning of the fifth paragraph of a term paper. After a bit of rewriting, he or she (take your pick) tries jumping to the marker, and is surprised to find out that it has "moved" into the middle of the fourth paragraph. What do you think may have caused this problem?

2. Another student summons the Editor to write a letter to the Editor. (I wonder if the Editor has an Editor?) He begins to type, but receives a "!" when he reaches the 80th column of the first line, and cannot seem to enter any further text. Help him out.

3. Yet another student prints a Chemistry lab report she has written with the Editor, and is surprised to see her pH table appear in paragraph form. Give her some friendly advice.

4. Why don't the S(et and M(argin commands appear in the Editor prompt line?

H. EXERCISES

1. Enter the Editor and set the Environment for word processing. Enter a single paragraph, then return to the Environment and set new margins; re-margin your paragraph. Repeat this procedure until you become familiar with it. Then enter a table, and try to margin it. Did you protect it? Finally, try to add a centered title at the top of your mini-report.

2. Create a form letter containing your resume and print it for three or four different companies. Hint: What portion(s) of the resume will need to be personalized for a particular company?

3. Write a novel (using the Editor as your word processor, of course). It should contain a meaningful plot, exhibit strong character development, and espouse a non-traditional point of view. This assignment will be due tomorrow, and must include a dedication, preface, and table of contents.

Chapter 21:
BELLS AND WHISTLES

This last lesson is a collection of miscellaneous items ("bells and whistles") offered in Apple FORTRAN that have not yet been discussed. It is designed to satisfy the thirsts of those who have to know *everything.* The information contained in this chapter was not withheld for sinister reasons, but merely to prevent an information overload (dare I say a STACK OVERFLOW?).

A. ALTERNATE WAYS OF SPECIFYING FORMATS

A Format can be referenced in four different ways:

1. `WRITE (unit number, format line number) output list`

or

`READ (unit number, format line number) variable(s)`

This is the familiar form we've used from the beginning.

2. `ASSIGN line number TO integer variable`

This special statement allows one to specify a *variable* (as opposed to a line number) in a READ or WRITE. The line referenced by ASSIGN may occur anywhere in the program (not necessarily before ASSIGN). Here's a sample use:

```
      ASSIGN 10 TO OUTPUT
      :
10    FORMAT (A, 3X, I3)

C     Reference Format by name, not by number
      WRITE (*, OUTPUT) NAME, NUMBER
```

REMEMBER: The variable in ASSIGN must be an Integer variable. In the above example "OUTPUT" would have to have been declared as such earlier in the program.

This statement may also be used to create descriptive GOTOs:

```
ASSIGN 3Ø TO MENU
ASSIGN 7Ø TO LOOP

:
IF (REPLY .EQ. 'Y') THEN
   GOTO LOOP
ELSE
   GOTO MENU
ENDIF
```

Strangely enough, the Computed (multi-option) GOTO will not allow variables in place of line numbers, even if ASSIGN has been used.

3. `WRITE (unit number, '(format specifiers)') output list`

or

`READ (unit number, '(format specifiers) ') variable(s)`

One may place the Format specifiers directly in the associated READ or WRITE. This method can help make programs more readable (you don't have to scan the entire program to find the FORMAT line), but is disadvantageous when the same FORMAT must be repeated several times. Here is a sample use:

```
WRITE (*, '(A, 3X, A)') CITY, STATE
```

4. The above methods share a common disadvantage: the *programmer* must specify the FORMAT within the program. Ideally, the *program user* could supply the FORMAT desired at execution time. Here is how he or she may do so:

```
        CHARACTER OUTPUT*3Ø

        :
        WRITE (*,1Ø) 'Output format (enclose in parentheses) ? '
1Ø      FORMAT (/, A, $)
C       Accept Format from program user
```

```
        READ (*, 2Ø) OUTPUT
2Ø      FORMAT (A3Ø)

C       Use the newly accepted Format specifier list
        WRITE (*, OUTPUT) STUDNT, COURSE, GRADE
```

Note that the variable used to read the FORMAT is a *Character variable. This is because we are now storing the Format specifiers themselves, NOT the FORMAT line number.*

Wise programmers can save themselves loads of Compiling/Linking time by using this method, especially when experimenting with carefully formatted output such as tables, charts, etc.

B. OVERLAYING LARGE FILES

Let's say you've written a rather large program that accesses ten subroutines from your library. Tacking all ten onto your Main Program would waste valuable memory space, or perhaps overflow memory outright. A more sensible approach would be to create ten overlays, thereby insuring that the subroutines would be loaded only when being used. (Recall that the Compiler and Editor operate in this fashion.) Here is how to do so:

```
$       USES USORT IN FORT1:SORT.CODE OVERLAY
$       USES UTABLE IN FORT1:TABLE.CODE OVERLAY
$       USES UMUSIC IN FORT1:MUSIC.CODE OVERLAY
etc.
```

This is the familiar "$USES" Compiler Directive. All you add is "OVERLAY" after the file name. Note that the files named as overlays must still be linked into the Main Program. (See Chapter 15 if you've forgotten how to do so.) In other words, the overlays are still part of the Code file, they just aren't all brought into memory at the same time.

C. LOGICAL VARIABLES

Apple FORTRAN supports Logical variables (Spock fans, take note). These variables have values of either ".TRUE." or ".FALSE.", and require *two bytes* of storage. Following is a (very) brief treatment of how they operate:

1. Declaration

The declaration of Logical variables requires a specification statement

called LOGICAL (logically enough). It is identical in form to the REAL, CHARACTER, and INTEGER statements. An example:

```
LOGICAL P, Q, FIXED, FOUND, DONE
```

2. Assignments

Take a look at the following examples:

STATEMENT	STORED
P = (1Ø .GT. 8)	.TRUE.
Q = (4 .LT. 1)	.FALSE.
FIXED = .FALSE.	.FALSE.
FOUND = (NAME .EQ. SEARCH)	?
DONE = (REPLY .EQ. 'Q')	?

Note how the right side of the assignment may be either a Logical *constant* (.TRUE. or .FALSE.) or a Logical *expression* (just as you would have following an IF). Also notice that some values may be determined only during the actual program execution.

3. Input of Values for Logical Variables

The input Format specifier is "L1". The field width of "1" is due to the fact that the only legal input is "T" or "F". Check out this example:

```
        WRITE (*, 1Ø) 'Logical value for BUSY (T/F)? '
1Ø      FORMAT (/, A, $)
        READ (*, 2Ø) BUSY
2Ø      FORMAT (L1)
```

4. Output of Logical Variables

The output Format specifier is "L1", and will yield either "T" or "F".

5. Conditionals

Decisions using Logical variables tend to throw people at first since there is no formal comparison of two values. However, *remember that a Logical variable is ALREADY true or false.*

STATEMENT	COMMENTS
IF (FOUND) GOTO 5Ø	if FOUND has value ".TRUE.", then goto 50

STATEMENT	COMMENTS
`IF (.NOT. DONE) GOTO 5`	".NOT." takes the opposite value of DONE; it changes ".TRUE." to ".FALSE." and vice versa
`IF (P .AND. Q) GOTO 3Ø`	goto 30 only if *both* have the value ".TRUE."
`IF (P .OR. Q) GOTO 3Ø`	goto 30 if *either* is ".TRUE."

Logical variables, especially in combination with descriptive GOTOs and subprogram names, can do much for FORTRAN's expressivity:

```
IF (FOUND) GOTO REPORT


IF (.NOT. USED) CALL UPDATE
```

D. THE "COMMON" STATEMENT

The COMMON statement provides another way to pass parameters back and forth between program units, and has this form:

```
COMMON / common block name / variable(s)
```

Following are two equivalent methods of using global variables:

METHOD 1

```
PROGRAM PHONE
COMMON / ONE / MOM, DAD, CHLDRN

:
CALL HOME

:
END

SUBROUTINE HOME
COMMON / ONE / MA, PA, KIDS

:
END
```

METHOD 2

```
PROGRAM PHONE

:
CALL HOME (MOM, DAD, CHLDRN)

:
END

SUBROUTINE HOME (MA, PA, KIDS)

:
:
END
```

COMMON is a specification statement, and as such, must be placed before any executable lines (on the same level as INTEGER, REAL, CHARACTER, LOGICAL, and DIMENSION).

You may use more than one Common block if you like, but each would require a different name. COMMON has one drawback: *it may contain either ALL Character variables or else NO Character variables (no mixing of character and numeric arguments).*

E. DATA AND FORMAT REPEAT FACTORS

These are best illustrated by example:

```
DIMENSION NUMBER (2Ø)
DATA NUMBER / 2Ø * Ø /
```

or

```
CHARACTER MESSGE*4Ø
DIMENSION MESSGE (1ØØ)
DATA MESSGE / 1ØØ * 'Enjoy birdwatching today and every day!' /
```

or even

```
DIMENSION VALUES (25)
DATA VALUES / 1Ø * -1.Ø, 5 * Ø.Ø, 1Ø * 1.Ø /
```

The above examples illustrate a fast way to initialize an entire array with the same value. As you have probably guessed, the "20 *" means "repeat the following 20 times".

Here is how to repeat Format specifiers:

OLD METHOD	NEW METHOD
`FORMAT (I5, I5, I5)`	`FORMAT (3 I5)`
`FORMAT (F6.2, F6.2, F4.1)`	`FORMAT (2 F6.2, F4.1)`
`FORMAT (A, I2, A, I2, A, I2)`	`FORMAT ( 3 (A, I2) )`

Note the nesting of Format specifiers in the last example.

F. HOLLERITH FORMAT SPECIFIER

This specifier was named after Herman Hollerith, the originator of the punched-card format (for the 1890 U.S. Census). Its form is:

```
number of characters H output characters
```

The following table should help give you the feel for Hollerith output:

WRITE (*, 10)		OUTPUT
10	FORMAT (3 HABC)	ABC
10	FORMAT (1 H%)	%
10	FORMAT (5 HBOB'S)	BOB'S

G. MORE APPLESTUFF

The APPLESTUFF library unit contains three routines we haven't yet discussed:

1. PADDLE

PADDLE is an Integer function that reads the game paddle positions, with this form:

```
PADDLE (paddle number)
```

The paddles are numbered 0 through 3. The value returned by PADDLE will be in the range 0–255.

Let's try a simple program to read paddle 0:

```
$       USES APPLESTUFF

        PROGRAM READIT
        CHARACTER REPLY*1

5       WRITE (*, 10) 'You are at position ', PADDLE (0)
10      FORMAT (/, A, I3)
        WRITE (*, 20) 'Another reading (Y/N)? '
20      FORMAT (/, A, $)
        READ (*, 30) REPLY
30      FORMAT (A1)
        IF (REPLY .EQ. 'Y') GOTO 5

        END
```

2. BUTTON

BUTTON is a Logical function that returns a value of ".TRUE." if the listed game paddle button is pressed. (*APPLE IIe and IIc users note: The*

OPEN-APPLE button corresponds to the paddle 0 button, CLOSED-APPLE to button 1.) This is the form:

```
BUTTON (game paddle button to read)
```

The program to read game paddle 0 could be rewritten with the help of BUTTON:

```
$       USES APPLESTUFF

        PROGRAM READIT
        INTEGER DELAY

        WRITE (*, 1Ø) 'Hold down button Ø to escape'
1Ø      FORMAT (A)

2Ø      WRITE (*, 3Ø) 'You are at position ', PADDLE (Ø)
3Ø      FORMAT (/, A, I3)

C       Delay loop so we have a chance to read the output!
        DO 4Ø DELAY = 1, 2ØØØ
4Ø      CONTINUE

        IF ( .NOT. BUTTON (Ø) ) GOTO 2Ø

        END
```

3. KEYPRE

The KEYPRE(ss) Logical function returns a value of ".TRUE". if any keyboard key has been pressed. It has been my experience that this function *ALWAYS returns a value of ".TRUE.", even if a key has NOT been pressed.* (I've often wondered whether or not this has something to do with the operating system's buffer.) If you want to try it anyway, a typical application would be:

```
C       Delay until a key is pressed
1ØØ     IF ( .NOT. KEYPRE() ) GOTO 1ØØ
```

H. MORE TURTLEGRAPHICS

1. SCREEN

The SCREEN Logical function returns a value of ".TRUE." if a coordinate is non-black, and ".FALSE." if it is black. Its form is:

```
SCREEN (X coordinate, Y coordinate)
```

2. DRAWBL

TURTLEGRAPHICS affords a means of creating image designs which is much more straightforward than Applesoft's legendary shape tables. In theory, one creates a two-dimensional Logical array, where a value of ".TRUE." represents a pixel that is non-black, and ".FALSE." represents one that is black. He or she then copies this array (or any portion thereof) onto the graphics screen with the DRAWBL (drawblock) subroutine. I use the words "in theory" because this feature is loaded with crippling bugs, none of which I've ever been able to overcome.

This is most unfortunate because, among other things, one could simulate a graphics screen dump to a disk file (there is no such system command available in Apple FORTRAN) in the following manner: SCREEN could be used to copy portions of the graphics screen into an array, which could be stored on disk; the file could then be retrieved at a later time, and redrawn with DRAWBL.

If you're a tinkerer and don't mind a bit of frustration, I hereby refer you to page 129 of the *Apple FORTRAN Language Reference Manual.*

I. ELSEIF

Apple FORTRAN supports an "ELSEIF. . .THEN" statement. It can be used with the IF. . .THEN. . .ELSE structure to code complex conditionals in an easy-to-read (and debug) form. Here is a sample use using Logical variables:

```
IF (LARGE .AND. LIGHT) THEN
   CALL SHRINK
   CALL DARKEN

ELSEIF (LARGE .AND. DARK) THEN
   CALL SHRINK
   CALL LIGHTN
```

```
ELSEIF (SMALL .AND. LIGHT) THEN
   CALL ENLRGE
   CALL DARKEN

ELSE
   CALL ENLRGE
   CALL LIGHTN

ENDIF
```

The above example is really one large test!

J. THE "BN" FORMAT SPECIFIER

Could a FORTRAN programmer produce software that a FORTRAN "illiterate" could use? After all, it is difficult for those who *do* know FORTRAN not to make an occasional mistake when inputting numeric data. It is *certain* that errors will occur for those who've never heard of "I4" and "F5.1".

Fortunately, Apple FORTRAN offers the "forgiving" Format specifier pair "BN / BZ", which stands for "Blanks read as Nulls / Blanks read as Zeroes". The default setting is "BZ", which results in the familiar behavior of READ statements we've come to know and love (?). By including "BN" in an input FORMAT, all blanks will be read as nulls (i.e., ignored). The "BN" specifier must be repeated in every FORMAT you wish to have it applied to; in other words, it governs only a *single* FORMAT statement. Let's see exactly how both specifiers behave. The **emphasized** leftmost column below represents the program user's typed reply, whereas the other columns show how the reply is interpreted by two different FORMATs:

```
1Ø      FORMAT (I4)        (note: "BZ" default)
2Ø      FORMAT (BN, I4)
```

IIII	READ (*, 1Ø)	READ (*, 2Ø)
24	24	24
1Ø3	1Ø3Ø	1Ø3
4 5	4Ø5	45
6	6ØØØ	6
123 4	123Ø	123
258	2	2

Notice that, with "BN", blanks are literally ignored; typing the spacebar is exactly the same as typing nothing at all. The only restriction still placed

upon input is that *it must occur within the specified field width to be accepted* (as the last two examples attempt to show).

At this point, I am always asked by a classroom full of angry students, "Why did you make us labor through all those input Formats if we could've just used BN?". My reply is, "Because many versions of FORTRAN offer no such convenience, and to learn FORTRAN without learning input FORMATs would be to not learn FORTRAN!" In addition, a solid understanding of input FORMATs was necessary to fully understand Formatted Data files, one of the most useful of all FORTRAN applications.

K. MISCELLANEOUS STATEMENTS

1. EXTERNAL function name

This statement is used when a programmer wishes to use the name of an *Intrinsic* function (such as SQRT, SIN, etc.) as the name of a *Subprogram* function. It looks like this:

```
EXTERNAL COS
```

"COS" now refers to a programmer-supplied Subprogram function, *not* to the System Library-supplied Intrinsic function.

2. EQUIVALENCE (variable, variable)

This statement will give *one* memory location *two names.* Here is a sample use:

```
EQUIVALENCE (DOG, MUTT)
```

Any reference to "DOG" is now a reference to "MUTT", and vice versa. This statement can be used when two different authors piece together a program, and each happens to use different names for the same variables. Note that the operating system does an *automatic* equivalence for the global variables (i.e., arguments) listed in subroutine/function calls and definitions.

EQUIVALENCE and EXTERNAL are specification statements, so let's recap the complete specification statement ordering list:

```
level one   : PROGRAM

level two   : IMPLICIT

level three : CHARACTER, COMMON, DIMENSION, EQUIVALENCE,
              EXTERNAL, INTEGER, LOGICAL, REAL
```

```
level four  : DATA

level five  : statement function definitions
```

3. STOP

This is a substitute "END", in that it causes the program to end, *but does not turn off compilation.* The following is a legal conditional:

```
IF (REPLY .EQ. 'N') STOP
```

When "STOP" is executed, you will see this message on the screen:

```
STOP

PROGRAM TERMINATED
```

4. PAUSE

Halts program control until RETURN is pressed. PAUSE may be placed anywhere in a program (as an independent statement), and will automatically yield this output:

```
PAUSE

PLEASE PRESS <SPACEBAR> or <CR> TO CONTINUE
```

L. PLACING LIBRARY FILES IN THE SYSTEM LIBRARY

This can be very useful, for when one does so and wishes to incorporate such a unit in a new program, he or she may type "R" for "RUN" instead of having to manually Compile and Link as described in an earlier chapter (although the "$USES" statement is still needed). There are, however, two drawbacks:

1. The process (by way of program FORT1:FORTLIB.CODE) is rather complicated, and not recommended for most users. (See pages 186–193 of the *Apple Pascal Operating System Reference Manual* for the documentation.)

2. The more units you add to the System Library, the less available space you have on your boot diskette; that space is needed to store your workfiles.

M. ADDITIONAL COMMAND LEVEL OPTIONS

Type "?" while at Command Level and you will see five new prompts:

1. USER RESTART

This will kick in the Command Level option most recently chosen. It is most useful to X(ecute the program you have just finished running, which this command will do without your needing to answer the "EXECUTE WHAT FILE ?" prompt.

2. INITIALIZE

Does a "warm boot" of the system. Some parameters such as the Prefix (default disk name) will not be "forgotten."

3. HALT

Does a "cold boot." This command is exactly the same as turning the power off, then back on again.

4. SWAP

The swapping option allows the operating system to handle larger programs than it normally could (by swapping *additional* overlays in and out of memory as needed). With swapping engaged, the Editor can handle files up to 19,968 bytes (39 blocks) in length, as compared to 17,408 bytes (34 blocks) in normal use. In addition, the Compiler will have approximately 12,644 words (25,288 bytes) available for symbol-table storage, versus 11,527 (23,054 bytes) without swapping. (*Technical aside: The figures for the Compiler were arrived at by using a dummy program consisting of only "PROGRAM" and "END". In real use, the figures reported would be lower, for they are affected by the number of specification statements and library units used.*) Surprisingly, compiling time is not affected by utilizing the swapping feature!

To implement swapping, type "**S**" from Command Level, and you will see:

```
SWAPPING IS OFF

TOGGLE SWAPPING?
```

"Toggle" means "switch from Off to On, or from On to Off", so type "**Y**" and you're in business. Those using operating system version 1.2 will have to choose a swapping level of 0, 1, or 2. The higher the number chosen, the greater the amount of swapping incorporated.

5. MAKE EXEC

EXEC files contain keystroke sequences that must be repeated often, or that may be extremely time-consuming (and you have something better to do than sit at the keyboard answering prompts). They help make possible such tasks as automated file printout and automated word processing. To create such a file, refer to pages 1–7 in the Addendum to the *Apple Pascal Operating System Reference Manual.*

N. AUTOMATIC PROGRAM EXECUTION

A file on the boot diskette with the special filename SYSTEM.STARTUP will automatically be executed when FORTRAN is booted. Our major problem with this kind of turnkey program is that we've already used such a program on FORT1 (to initialize FORTRAN for the use of APPLESTUFF and TURTLEGRAPHICS, remember?). If you'll never be using these library units, you may delete the old SYSTEM.STARTUP and add your own.

O. REVIEW QUESTIONS

1. Who is your favorite FORTRAN author?

2. What is the worst pun contained in this book?

P. EXERCISES

1. Do fifty jumping jacks, twenty-five sit-ups, and twenty-five push-ups (I've been waiting twenty-one chapters to get this one in!).

2. The features that will be of most use from this chapter are the ASSIGN statement, Logical variables, and the "BN" Format specifier. Select one of your recently-written programs, and spruce it up with one or more of the above. Perhaps you can also find some use for a DATA or FORMAT repeat factor, an ELSEIF. . .THEN, a PAUSE, or some of the other topics we've just covered.

3. If you still have a program on disk that incorporates Code library files, try making the Main Program call them as overlays. Pass your arguments to the program units in a COMMON statement. C(ompile, L(ink, and X(ecute this new version.

4. Write a program to repeatedly read Paddle 0 in the range 0 through 50, and play the corresponding NOTE with a duration of 1 *only if Button 0 (Open-Apple) is being pressed.* In a sense, you're letting the program user control pitch with the paddle (s/he can turn it up to get higher notes, and down to get lower notes) and duration with the button (the longer it's held down, the longer the note plays). When Button 1 (Closed-Apple) is pressed, end the program.

NOTE: Although the MOD function with a divisor of 51 would be useful for limiting PADDLE's 0–255 range, it would produce this pattern of readings: 0, 1, 2, . . . , 48, 49, 50, 0 (!), 1, 2, . . . , 48, 49, 50, 0, 1, 2, etc. It would be better to select fifty-one *intervals,* where five or so *consecutive* readings produced note 0, the next five produced note 1, and so on.

5. Write a program to allow the game paddles to draw on the TURTLEGRAPHICS screen. Let Paddle 0 represent the X coordinate, and Paddle 1 the Y coordinate. Button 0 could be pressed to change colors, while Button 1 could be the QUIT signal. Again, you should *not* use MOD to convert the paddle readings into the appropriate ranges.

APPENDICES

Appendix A:
ANSWERS TO REVIEW QUESTIONS

CHAPTER 1

1. An overlay is one of the segments into which a very large program has been divided so that it may fit into the available memory. Since the Editor contains overlays, one must leave disk FORT2 in at all times when editing. This allows different overlays to be called in as needed.

2. If no file was read in, one presses the <**ESC**> and <**RETURN**> keys when the "NO WORKFILE IS PRESENT. FILE ?" prompt appears. See question 7 for recovery when an old workfile has been read in.

3. disk name : filename . suffix

4. /R/WISCONSIN//MINNESOTA/ (Note the infinite repeat factor "/" typed *before* the "R".)

5. Up-arrow, down-arrow, left-arrow, right-arrow, <TAB>, <RETURN>, and P.

6. A workfile is a scratchpad, temporary copy of your developing program. It is created when you enter a program into the Editor, then choose to **Q**(UIT THE EDITOR, and **U**(PDATE THE WORKFILE. This option saves the newly written program with the special filename FORT1: SYSTEM.WRK.TEXT.

7. One chooses to **Q**(UIT THE EDITOR and **E**(XIT WITHOUT UPDATING. In our next chapter, we'll discuss how to use the Filer to remove the old workfile.

CHAPTER 2

1. **E**(dit
replace disk FORT1 with disk GAS
Prompt—NO WORKFILE IS PRESENT. FILE ? **Reply**—GAS:MILEAGE

replace disk GAS with FORT1
Q(uit the Editor
U(pdate the workfile

2. **Q**(uit the Editor
E(xit without updating
F(iler
N(ew workfile
Prompt—THROW AWAY CURRENT WORKFILE? Reply—Y
Q(uit the Filer
E(dit

3. **T**(ransfer
Prompt—TRANSFER ? Reply—WINDY:=CODE
Prompt—TO WHERE ? Reply—DRASTIC:$

4. It allows one to change the understood disk name (normally FORT1:). If you'll be using your disk name often, changing the Prefix name to that of your disk will save a lot of unnecessary typing.

5. ? prompts yes or no for each file contained on the specified disk
= wildcard (any characters) symbol
$ same name
volume numbers allow a single number (as opposed to a word) to represent an operating system device (i.e., **#6:** instead of **PRINTER:**).

6. **a.** he may have forgotten to specify his disk name (so the Filer looked on disk FORT1)
b. his disk may not have been in one of the two drives
c. he may have forgotten the ".TEXT" suffix
d. he may have misspelled some other part of the filename.

7. He would wind up with files ARMY:FLAG.TEXT.TEXT and ARMY:FLAG.TEXT.CODE. He would then have to use the C(hange command to rename the files.

8. **R**(emove
Prompt—REMOVE ? Reply—DISK:=

CHAPTER 3

1. It would probably be wiser to let the Compiler continue on the initial attempt so you may find *all* the mistakes. (Geenen's law has a corollary: The number of Compile-time errors is proportional to program size.) Once you have returned to the Editor and attempted to correct them, you may recompile. At this stage there should be far fewer mistakes, so you may now return to the Editor for each.

2. Once the Editor has the workfile loaded, type "**34<RETURN>**", which will bring you to the start of the 34th line. Upon correcting it, type "**22<RETURN>**" to get to line 56, then "**30<RETURN>**" to find line 86. Repeat factors are surely a blessing!

CHAPTER 4

One of the programs to be linked could be a useful library program that had been written and compiled earlier. Linking it into a new program eliminates the need to type it and compile again.

In addition, program modules may be developed independently (perhaps even by different persons) and linked together upon completion to form a working whole.

CHAPTER 5

1. The *Code* version is saved because it is the executable (P-code) version. Code files cannot be edited, nor are they human-readable, so the *Text* version is saved to allow future changes and to provide documentation.

2. Editor—workfile read in automatically
Run option—automatically compiles, links, and executes workfile
Compiler—two prompts automatically answered
Linker—all five prompts automatically answered
Filer—single command saves both Text and Code versions of workfile

In addition to eliminating the need to answer nearly all prompts, workfiles also eliminate the need to shuffle disks in and out of the drives during program development.

3. The Editor and the Compiler.

4. The Compiler would be summoned (ready for the workfile), followed by the Linker, before anything could be executed.

CHAPTER 6

1. Columns 1–5 are reserved for line numbers. They should contain spaces if there is no line number. Column 6 contains an asterisk if the current line is a continuation of the previous line, otherwise it too is blank. Columns 7–72 contain the program instructions; blanks are not significant in these columns (unless part of a character string), so programmers are free to use indentation and spacing to achieve greater readability. Columns 73–80 may contain text, but they will not be read by the Compiler.

2. Integer and Real variables are determined by default. Variable names beginning with letters I through N are understood to be Integer, while variable names beginning with letters A through H, or O through Z are understood to be Real. Character variables, on the other hand, must be declared in a CHARACTER statement.

3. The first line must be the PROGRAM statement, while the last line must be the END statement (followed by a single press of the RETURN key).

4. The DATA statement allows one to set initial values for several variables in a single line. It is placed after the CHARACTER statement because a variable must have been declared as being of type Character before it may be assigned a Character value in DATA.

5. a. AREA is of type Real, so the value it stores is 6.0 (the Real equivalent of Integer 6).

b. INDEX is of type Integer, so the value it stores is 28 (note that the decimal is truncated, *not* rounded).

c. The expression 4.0 / 2 is mixed (one Real, one Integer) so it returns the Real value 2.0, which is then stored in Real variable VALUE without modification.

d. The expression 4 * 3.0 is performed first since it is in parentheses. This mixed-type expression yields an answer of 12.0 (Real), which is then raised to the second power. The base 12.0 is Real, while the exponent 2 is Integer, so the value returned, 144.0, is Real. Since ANSWER is a Real variable, 144.0 is stored without alteration.

e. "*" and "/" have equal precedence, so we work from left to right.

Note that all right-hand side values are Real, so all operations yield Real results. 4.0 * 2.0 yields an answer of 8.0, which is then divided by 10.0. Thus the right-side expression simplifies to 0.8. LENGTH is an Integer variable, so the value it stores is 0, because of the application of the INT truncation function.

f. 27 / 4 would normally result in a value of 6.75 if at least one of the operands were of type Real. But since both are of type Integer, it evaluates to 6. Since FINAL is a Real variable, the REAL function is automatically applied, so the value stored is 6.0.

6. The Compiler would read the apostrophe as a single quote, which would close off the opening single quote after the word "LET". The additional characters would, therefore, not be expected, and would cause an "Unrecognizable Statement" error.

CHAPTER 7

No questions.

CHAPTER 8

1. a. ␣-46.8
b. 128.3␣␣
c. ****
d. ␣28
e. ␣␣␣6
f. **
g. Triangle
h. ␣Rectangle
i. Penta
j. ␣␣␣Hexagon␣␣

2. a. 3␣
b. 6␣␣␣
c. 2
d. 2.6
e. 28.6
f. 249.␣ (technically "249.")
g. 'Single'
h. 'Double␣␣␣␣'
i. 'T'

3. TEMP is a Real variable, but the programmer is trying to WRITE its value with an "I" FORMAT.

CHAPTER 9

1. If she chose the fifth option, the Computed GOTO would be ignored since it has no fifth line number. Since it is a wise programming practice to place an error-handling routine after the Computed GOTO, option 5 would be treated as an invalid reply.

2. The word "THEN" serves to flag a conditional of the Block (multiple-statement) type, whereas if "THEN" does not appear, a Logical (single-instruction) conditional is expected.

3. When it contains only yes-branch instructions.

4. No error message appears. The smaller Character string is padded with trailing blanks, then compared to the larger string.

CHAPTER 10

1. The Compiler will not confuse the adjacent decimal points (for the numbers) with the periods (for the logical operators). Even so, zeroes should be added for readability; in fact, the conditional is poorly stated and should be changed to:

```
IF (X .GT. Ø.5  .AND.  X .LT. 1.Ø) etc...
```

2. Even though the statement is redundant (since NUMBER is already an Integer variable), no error will be reported.

3.
a. Character
b. Character
c. Real
d. Integer
e. Integer
f. Real
g. Real
h. Integer
i. Real
j. Integer
k. Real
l. Real

CHAPTER 11

1. Since the program device number is independent of the operating system device number, the printer could be assigned program device 8. Even so, why not keep them the same?

2. The loop would be ignored, unless a negative step was specified.

3.
```
INDEX = 11
NUMBER = 31
INDEX = Ø
NUMBER = 8
SUBSCR = LAST + 1
COUNTR = LAST + STEP
```

4. It would perhaps be easiest to write such a loop *without* using DO. If we assume "INDEX" has been declared as a Real variable, here is an example:

```
C       Loop running from Ø to 5 with increment of Ø.5
        INDEX = Ø.Ø

5Ø      IF (INDEX .LE. 5.Ø) THEN
           :
           loop instructions
           :
           INDEX = INDEX + Ø.5
           GOTO 5Ø
        ENDIF
```

CHAPTER 12

```
INTEGER VALUE
CHARACTER SUIT*8
DIMENSION VALUE(13), SUIT(4)
```

CHAPTER 13

1. Specification statements are the first lines listed in a FORTRAN program. They name the program (PROGRAM), set default variable types (IMPLICIT, INTEGER, REAL, CHARACTER), define arrays (DIMENSION), assign initial values for variables (DATA), and define Statement (single-line) functions. In short, they set up the specifications for the actual program which follows.

2. The Specification statements are in the wrong order. IMPLICIT should be immediately after PROGRAM, followed by CHARACTER and DIMENSION. (Technically, DIMENSION could also precede CHARAC-

TER.) Finally, the line number "10" should be placed on the first executable (non-specification) line. Program control may not return to the PROGRAM statement, since it is a specification statement.

3. **a.** 10.0
   b. –23.0
   c. 42
   d. error; NEWSUM expects three INTEGER values!
   e. 19.7
   f. 12.8

CHAPTER 14

1. **a.** NUMBER OF VALUES RETURNED TO MAIN PROGRAM : Function—normally only one; Subroutine—any number from zero to over 100.
   b. METHOD FOR RETURNING VALUE(S) : Function—value returned as Function name; Subroutine—values returned are those listed as arguments.

2. **a.** METHOD OF COMPILATION : both functions and subroutines are compiled independently of the Main Program.
   b. METHOD OF RECEIVING MAIN PROGRAM VARIABLES : both receive only those variables listed as arguments, which are global; all other variables are local.
   c. USE IN STRUCTURING PROGRAMMING TASKS : both allow the programmer to divide a large program into appropriate subtasks, each of which may be developed independently as a function or a subroutine.

3. Global variables have values which may be obtained in two compilation units (Main Program and either a function or a subroutine). If either of these compilation units alters the value of a global variable, the change will also be reflected in the other unit.

Local variables have values which are not known outside the compilation unit (Main Program, function, or subroutine). They may be altered only from within a single compilation unit.

All variables (and line numbers, for that matter) are local to the compilation unit unless they are listed as arguments in a function or subroutine call.

4. Since these units may be developed independently (at different times, or even by different persons), there is no way to know which line numbes will and will not be used in the other unit(s). Line numbers must be local in order to prevent "clashes," which occur when the same line number is used more than once.

5. Again, since unit development is independent, one cannot always know which variable names to choose so that they will be the same in each compilation unit. Hopefully, the names will at least be synonymous (TESTS and EXAMS, for instance).

6. Yes, it is possible. Subroutines are often "self-starting"; once they are called, they may proceed on their own without receiving any values from the Main Program. (You may want to look at the FORMAT program given at the end of Chapter 8 to see an example.) No parentheses are necessary after the subroutine name when you are defining or calling a subroutine without arguments.

7. The "*" allows an array to have an assumed-size dimension (determined by the dimension of the corresponding array from the main program). This feature allows the assumed-size array to work with arrays of varying sizes.

CHAPTER 15

1. A buffer is a portion of memory whose contents change with each deletion or insertion. These contents may be copied back into the Editor at any position the programmer desires. It is most valuable for moving text from one portion of the program to another.

2. **a.** One may copy *Text* library files into the Editor with the C(OPY F(ROM FILE option.

b. One may incorporate *Text* library files during compilation with the $INCLUDE Compiler Directive.

c. One may include *Code* (i.e., already compiled) library files with the Linker, provided the Main Program issued a $USES Compiler Directive.

3. FORT2 contains the Editor, which is written with overlays. This requires FORT2 to be present at all times while editing. Therefore, one must remove FORT1 from Drive 1, so the disk containing the library file may be read.

4. Disk FORT1 contains the workfile SYSTEM.WRK.TEXT. Disk FORT2 contains the Compiler, which is written with overlays, so neither system disk may be removed while compiling a workfile. The library file must, therefore, be transferred to FORT1 before compilation begins (FORT2 is write-protected, in case you've forgotten).

5. When the Compiler receives the $USES Directive, it checks to make sure that the named library file is on-line. So, for reasons similar to those given in number 4, the library file must be transferred to disk FORT1 before compilation. This also eliminates the need to shuffle disks when the library file is actually linked.

6. One type of text which lends itself especially well to use in this manner is a library of *Statement* functions. (Recall that "$INCLUDE" cannot read a *Subprogram* function.) As a typical example, you could develop a library of ten or fifteen Statement functions to do metric conversions. Then there would no longer be a need for you to re-write these useful functions, since you may simply incorporate a "`$INCLUDE FORT1:CONVERSION.TEXT`" in future programs.

Another application would be with a rather large list of specification statements. If the same extensive "DATA" statements would need to be used in several programs, for instance, they may be written *once*, stored in a diskfile, then "$INCLUDEd" into each separate program.

CHAPTER 16

1. A Formatted Sequential file consists of multiple records, separated from each other by the RETURN character. These records in turn consist of fields, which are adjacent (i.e., not separated from each other). The number of fields contained in each record may vary. The Formatted Sequential file structure contains no padding.

2. READ and WRITE move the pointer forward; BACKSPACE moves it back one record, while REWIND and OPEN move it back to the beginning of the file.

3. Since we are now using external files, our READs and WRITEs will have unit numbers other than "*". If we used "0" for specifying the Console, it could not be easily seen which READs and WRITEs are for files and which are for the Console. The asterisk is a visual aid, as I see it (pardon the pun).

4. Yes; for instance

```
OPEN (4, FILE = '#4:ADDRESS.DATA')
```

will open a file on *any* disk in drive volume #4 (Drive 1).

5. If you choose device numbers 4 and 5 for files, and 6 for the printer, your *program* device numbers will be the same as the *operating system* device numbers. (Why choose different numbers and risk confusion?)

CHAPTER 17

Unformatted Sequential Files

Unformatted Sequential files simply contain one field after another, with no separators; there are no records. Formatted Sequential files do have records, which are separated by RETURNs; these are then broken down into adjacent fields.

Unformatted OPENs contain a "`FORM = 'UNFORMATTED'`" parameter, which of course does not appear in Formatted OPENs.

Unformatted READs and WRITEs contain no FORMAT line number; Formatted READs and WRITEs do.

Unformatted files may not be BACKSPACEd, while Formatted files may.

All other commands are identical.

Unformatted Direct-access Files

1. The variable type being used to read the field determines the number of bytes to move the file pointer (2 for an Integer variable, 4 for a Real variable, and 1 for each byte declared in a Character variable).

2. 3 + 2 + 2 + 2 + 4 = 13 bytes. (I hope you're not superstitious).

CHAPTER 18

1. This problem arises when FORT1 is not on-line. The SYSTEM.LIBRARY (and therefore APPLESTUFF) cannot be accessed, so the system hangs.

2.
```
GUESS = 75 + MOD (RANDOM ( ), 26)
```

3. NUMBER's values would be in the range −5 through +5, inclusive.

4. Unfortunately, harmonies and chords are not possible, since the NOTE subroutine has only one voice. However, if you'd like to sing along. . . .

CHAPTER 19

1. Because the INITTU subroutine calls in and *initializes* the graphics screen (the screen is erased, the turtle is positioned in the center, etc.), while GRAFMO brings in the graphics screen *intact* (the screen is not erased, the turtle is not re-positioned, etc.).

2. Surprisingly, only two! You simply set the bounds of the rectangle with VIEWPO, then use FILLSC to fill it (using color code 10).

3. Because MOVE (10) is relative to the turtle's location and direction. For instance, if you are at the lower left corner of the screen and the turtle's angle is 0 degrees, MOVE (10) will draw a horizontal line along the bottom edge of the screen. If you are at the upper right corner of the screen and the turtle angle is 270 degrees, MOVE (10) will draw a vertical line along the right edge of the screen. Get the picture? (Ouch!)

4. Although MOVETO (0, 0) is not affected by the turtle's angle, it is still relative to the turtle's current position. For example, if the turtle is at the lower right corner of the screen, MOVETO (0, 0) will produce a horizontal line along the bottom edge of the screen. If the turtle is at the upper right corner of the screen, MOVETO (0, 0) will produce a diagonal line from the upper right to the lower left corner.

CHAPTER 20

1. The problem is that a marker is actually an *absolute character position* in the file. In other words, when you set marker A, the Editor remembers that it is placed at, say, the 345th byte of the file. If you later insert text *before* that marker, the 345th byte will wind up "moving backwards," while a deletion between the marker and the beginning of the file will appear to cause the marker to move ahead.

2. The student must **S**(et the **E**(nvironment parameters **A**(uto Indent to **FALSE**, and **F**(illing to **TRUE**. He may then exit the Environment and begin word processing. Note that when Filling is FALSE, no word wrap-around will take place.

3. Tell this student to place the Command Character "^" in front of each line of her tables from now on. Assure her that they will protect these lines from the Margin command, but will *not* appear on the printed copy.

4. The Editor prompt line is already jammed with options. There's no room for two more! Since most users will probably not use the word processing capabilities of the Editor (which are limited in comparson with today's feature-packed word processors), S(et and M(argin are reasonable choices for omission. Although you may be thinking that the Editor could follow the Filer's lead and simply list *letters* on its prompt line, such a practice would undoubtedly confuse novice users ("Is R for Remove or Replace?", "Is D for Date or Delete?", etc.). In fact, I'd like to see the *Filer's* prompt line changed. (Note: This diatribe was written before the advent of operating system version 1.2.)

CHAPTER 21

1. Donald J. Geenen (and that's an *order!*)

2. They're *all* bad, but the one contained in Exercise 1 of this chapter is a contender.

Appendix B:
HELP WITH ERROR MESSAGES

This appendix will provide you with some help in understanding some of the more common error messages, many of which can be quite cryptic in the eyes of a novice Apple FORTRAN user. The source for the error message texts (which appear in capital letters below) was the *Apple FORTRAN Language Reference Manual.*

A. COMPILE-TIME ERRORS

1 FATAL ERROR READING SOURCE BLOCK : The file being compiled cannot be read correctly. Try a BAD BLOCKS scan. Perhaps you need a new disk.

2 NONNUMERIC CHARACTERS IN LABEL FIELD : The first five columns are known as the label field. They are reserved for line numbers (exceptions are a "C" in column one for a comment line, or a "$" in Column one for a Compiler Directive). If a line does not need a line number, these five columns must be blank.

3 TOO MANY CONTINUATION LINES : A program line may be followed by a limit of nine continuation lines.

4 FATAL END OF FILE ENCOUNTERED : A program must end with the END statement followed by a *single* press of the RETURN key. In addition, END may only be placed as the *last* line of a program.

5 LABELED CONTINUATION LINE : A continuation line may not have a line number. Simply leave Columns 1–5 blank, and place your asterisk in Column 6.

11 INTEGER CONSTANT OVERFLOW : Integers must be in the range −32,768 to +32,767.

12 ERROR IN REAL CONSTANT : Real numbers must have an absolute value in the range 1E−38 to 1E+38.

14 IDENTIFIER TOO LONG : Variables, as well as program, function, and subroutine names, may be no more than six characters in length.

15 CHARACTER CONSTANT EXTENDS TO END OF LINE : You are missing the end quote on a message. You must close any open quotes by column 72.

16 CHARACTER CONSTANT ZERO LENGTH : You must place a minimum of one character between the single quotes when assigning a value for a Character variable.

54 TYPES OF COMPARISONS MUST BE COMPATIBLE : You may not compare Real or Integer data with data of type Character.

61 ILLEGAL ASSIGNMENT - TYPES DO NOT MATCH : You may not assign Real or Integer values to a variable of type Character. Likewise, you may not assign data of type Character to a Real or Integer variable.

75 SUBSCRIPT OUT OF RANGE IN DATA STATEMENT: If you dimension an array at five elements for example, don't place more than five values in the DATA statement which initializes that array.

78 TYPE CONFLICT IN DATA STATEMENT : Real variables must receive Real values in DATA. (Don't forget to use a decimal.) Similarly, Character variables must be assigned Character values (don't forget the single quotes).

87 ERROR IN TYPE OF ARGUMENT TO AN INTRINSIC FUNCTION : A function anticipating a Real argument was sent an Integer, or vice versa.

89 UNRECOGNIZABLE STATEMENT : This is Apple FORTRAN's general, all-purpose "Syntax error" message. Check your spelling of keywords. Also, you may have used a comma when a period was expected, etc.

91 MISSING END STATEMENT : A program must end with the END statement followed by a single press of the RETURN key.

93 & 94 FEWER (or MORE) ACTUAL ARGUMENTS THAN FORMAL ARGUMENTS IN FUNCTION/SUBROUTINE CALL : If your subprogram is defined with three arguments, send it exactly three arguments, and so on.

95 TYPE OF ACTUAL ARGUMENT DOES NOT AGREE WITH TYPE OF FORMAL ARGUMENT : Your function or subroutine is expecting an Integer argument, but you are sending it a Real argument, etc.

96 THE FOLLOWING PROCEDURES WERE CALLED BUT NOT DEFINED : The Compiler is providing you with a list of functions and/or subroutines which the Main

Program was attempting to use, but that were not defined. The only way you may override this error message is to place the subprogram(s) below the END statement of the Main Program, or else use the "$USES" Compiler Directive to inform the Compiler that the subprogram(s) will be linked in *after* compilation.

1Ø3 LABEL ALREADY USED FOR FORMAT : Don't use 10 as a line number if you have also used it for a FORMAT statement.

1Ø4 LABEL ALREADY DEFINED : You have given two different lines the same line number.

1Ø5 JUMP TO FORMAT LABEL : Don't use "GOTO 20" if line 20 is a FORMAT statement.

131 LABEL REFERENCED BUT NOT DEFINED : You used "GOTO 30" but have no line 30, for example.

132 DO OR IF BLOCK NOT TERMINATED : You have used either a DO loop whose last line was not reached before the END statement (did you list the wrong line number in the DO?), or an IF. . .THEN. . .ELSE whose ENDIF was not reached before the END statement (did you forget the ENDIF?).

135 FORMAT MUST BE LABELED : Don't forget a line number for your FORMAT.

159 UNRECOGNIZABLE I/O UNIT : You attempted to WRITE to or READ from a device which does not exist. Most likely you have forgotten to declare the unit number in an "OPEN" statement.

161 OPTIONS EXPECTED AFTER "," IN I/O STATEMENT : Don't use a comma at the end of a WRITE or READ unless you have another item forthcoming.

163 LABEL USED AS FORMAT BUT NOT DEFINED IN FORMAT STATEMENT : You used "WRITE (*, 40)" but never defined a FORMAT line 40, for example.

165 LABEL OF AN EXECUTABLE STATEMENT USED AS A FORMAT : You used "WRITE (*, 50)" but line 50 is not a FORMAT statement, for example.

169 FUNCTION CALLS REQUIRE "()" : You must use a pair of dummy parentheses when calling a function with no arguments. (This is *not* true for subroutines without arguments, however.)

2Ø4 UNABLE TO OPEN $USES FILE : The file to be USED must be transferred to FORT1 before compiling.

208 THERE IS NO SUCH UNIT IN THIS $USES FILE : You may have misspelled the function or subroutine name, or you may have forgotten to place a "U" in front of it.

400 CODE FILE WRITE ERROR : The disk is either full, needs to be crunched, or is defective (try a BAD BLOCKS scan).

B. RUN-TIME ERRORS

602 SIGN NOT FOLLOWED BY DIGIT IN INPUT : Anytime you use a "+" or "−", you must follow it with at least one digit. This frequently occurs when a person tries to type something like "−5" in response to an "I1" FORMAT. Remember that the negative sign takes up one field character, too!

603 DIGIT EXPECTED IN INPUT : The program anticipated an Integer or Real value, but you typed a Character value.

616 X FIELD IN FORMAT REQUIRES REPETITION FACTOR : Even if you move only one space, you need a number in front of the "X" Format specifier.

630 ATTEMPT TO PERFORM I/O ON UNKNOWN UNIT NUMBER : You attempted to WRITE to or READ from a device which does not exist. You may have forgotten to declare the unit number in an "OPEN" statement or may have placed the READ or WRITE before the corresponding OPEN. Perhaps the file was open, but you inadvertently closed it.

633 I FORMAT EXPECTED FOR INTEGER READ : You used an Integer variable in READ, but failed to use the "I" Format specifier in the associated FORMAT.

634 F OR E FORMAT EXPECTED FOR REAL READ : You used a Real variable in READ, but failed to use an "F" or "E" Format specifier in the associated FORMAT.

640 A FORMAT EXPECTED FOR CHARACTER READ : You used a Character variable in READ, but failed to use the "A" Format specifier in the associated FORMAT.

641 I FORMAT EXPECTED FOR INTEGER WRITE : You used an Integer variable in a WRITE, but failed to use an "I" Format specifier in the associated FORMAT.

642 W FIELD NOT GREATER THAN D FIELD + 1 : The "F" Format specifier's field width must be at least one larger than the decimal field, since the decimal

itself also takes up one field character. Therefore, the following specifiers are illegal : F4.7, F5.6, and even F3.3.

644 E OR F FORMAT EXPECTED FOR REAL WRITE : You used a Real variable in a WRITE, but failed to use an "F" or "E" Format specifier in the associated FORMAT.

646 A FORMAT EXPECTED FOR CHARACTER WRITE : You used a Character variable in a WRITE, but failed to use an "A" Format specifier in the associated FORMAT.

655 ATTEMPT TO DO EXTERNAL I/O ON A UNIT BEYOND END OF FILE RECORD : You tried to READ or WRITE past the end-of-file character in a Data file.

1ØØØ+ COMPILER DEBUG ERROR MESSAGES : These are my favorites! The manual notes that these messages "should never appear in a correct program." I think it is generally true that *none* of the error messages should ever appear in a correct program! Luckily, I've never seen one of these rather strange error messages reported.

Appendix C:
DISKETTE SUPPLEMENT PROGRAMS

Name of disk - LAF:

REFERENCES

REFERENCES

[1] *Apple Pascal Operating System Reference Manual*

[2] *Apple FORTRAN Language Reference Manual*

[3] Boillot, Michel. *Understanding FORTRAN,* Second Edition. West Publishing Company, St. Paul, 1981.

[4] The author's four years of experience with Apple FORTRAN

INDEX

INDEX

DISKETTE ORDERING INFORMATION

THE LEARNING APPLE FORTRAN DISKETTE

The Learning Apple FORTRAN Diskette is available from the publisher and recommended for use with *Learning Apple FORTRAN.* This diskette contains text versions of all sample programs found in the book and, for many of the longer and more elaborate programs, the code (or compiled) versions as well. By eliminating the need for entering data, this diskette allows the user more time for learning.

COMPUTER SCIENCE PRESS DISKETTE POLICY

This diskette is not protected. With purchase of this diskette you are granted permission to make copies as needed for use in your school, but not for transfer between schools. Diskettes are not to be resold.

ORDER FORM

Ordering Information

Call (301) 251-9050 or write to Computer Science Press, Inc., 1803 Research Boulevard, Rockville, Maryland 20850 to order our publications. Ask for our complete catalog of quality books at all levels from introductory to the advanced levels. Residents of Maryland should add 5% sales tax. Prices subject to change without notice.

		QUAN.	PRICE
LEARNING APPLE FORTRAN DISKETTE Donald J. Geenen 0-88175-030-1	@ $17.00	_____	_____
	Subtotal		_____
	Postage and handling $2.00		_____
	Total		_____

☐ Payment enclosed ☐ VISA No. _____ ☐ MasterCard No. _____

Signature _____ _____ Expiration date _____

Name _____

Address _____

City _____ State _____ Zip _____

ALL ORDERS FROM INDIVIDUALS MUST BE PREPAID.

☐ Add my name to your mailing list. ☐ Send me your current catalog.

COMPUTER SCIENCE PRESS, INC., 1803 Research Boulevard, Rockville, MD 20850, USA - (301) 251-9050

ABOUT *LEARNING APPLE FORTRAN*

Learning Apple FORTRAN begins with a thorough coverage of the Apple FORTRAN Operating System. This operating system is very similar to the one used with Apple Pascal. After developing a mastery of the Apple FORTRAN Operating System, the book proceeds to spell out the syntax and form of 1977 ANSI FORTRAN with the help of numerous sample programs and program segments. Emphasis is placed on program readability and expressiveness.

Each chapter contains a set of review questions and hands-on exercises to reinforce the material as it is presented in the book. Appendices contain the answers to all review questions and provide explanations of the most common Compile-time and Run-time error messages.

ABOUT *THE LEARNING APPLE FORTRAN DISKETTE*

The Learning Apple FORTRAN Diskette is available from the publisher and recommended for use with this text. This diskette contains text versions of all the sample programs contained in the book and, for many of the longer and more elaborate programs, the code (or compiled) version as well. By eliminating the need for entering program data, this diskette allows more time for learning.

ABOUT THE AUTHOR

Donald J. Geenen attended Marquette University in Milwaukee, Wisconsin, and graduated *magna cum laude* in June, 1981, from Lawrence University of Appleton, Wisconsin, where he was elected to Phi Beta Kappa.

Donald Geenen has taught computer programming courses in BASIC, FORTRAN, Pascal, and word processing since 1981 at Premontre High School in Green Bay, Wisconsin.

ISBN 0-88175-024-7